Rheinisch–Westfälische Technische Hochschule Aachen
Aachen University
Department for Computer Graphics and Multimedia
Prof. Dr. Leif Kobbelt

Lecture Notes on

Computer Graphics I

Sandip Sar-Dessai
Mario Botsch

This document contains lecture notes on "Computer Graphics I" held by Prof. Dr. Leif Kobbelt during summer term 2003 at RWTH Aachen.

The lecture notes are mostly based on the videos recorded by Klaus Ridder and available at

`http://www.s-inf.de/video`

When necessary, lecture slides (available at the department's WWW-pages) and additional literature was sourced.

Special thanks go to Carsten Fuhs for reading the text and giving valuable advise with LaTeX. The lecture notes were verified by the department.

However, it is still quite probable that mistakes can be found in the notes. I would be very grateful if you mailed me further corrections, criticism or improvement proposals.

The current version of this document can be downloaded from the homepage of the department.

Sandip Sar Dessai
E-Mail: `sandip.sardessai@gmx.de`
Homepage: `http://www.sandip.de`

Department for Computer Graphics and Multimedia (I8)
Homepage: `http://www-i8.informatik.rwth-aachen.de`

1st edition, October 15th, 2003.
2nd edition, September 13th, 2004.
3rd edition, May 11th, 2005.

Contents

Chapter 1

Introduction

1.1 Computer Graphics

In this introductionary course on Computer Graphics we will be dealing with the generation and manipulation of virtual models, and with the rendering of these models in order to display them to the computer user. Since we want to build realistic virtual counterparts of real-world models and environments, we are mostly working on three-dimensional geometry that has to be displayed on a two-dimensional screen. The issues to be solved include the following:

- Modeling: in order to create highly detailed realistic models we need a mathematical toolkit to approximate these models using different geometric representations of surfaces or solids.
- Data structures and algorithms: we need a way to represent the scenes in terms of data structures, and we need efficient algorithms to work on them.
- Complexity and transmission: most of the scenes we are dealing with are quite complex, requiring large amounts of data. Therefore, we need to find ways to store the data efficiently in memory or on disc, and to transfer them through networks.
- Software design and management: doing all the theoretical framework would not make sense without actually implementing it. Thus, creating software is also part of Computer Graphics.

This course will introduce you to the basic concepts in Computer Graphics. More specifically, we will discuss the process of rendering 2D images from a 3D scene description. The course will be organized according to the different stages in this process (cf. Fig. 1.1):

1. Geometry: we will discuss how to represent and how to work with 3D objects.
2. Local Illumination: when rendering a 3D scene into a 2D image using local illumination we neglect global effects, i.e. effects that are caused by the interaction of objects with each other (such as shadows and reflections). The big advantage of this rendering mode is speed: all local effects can be calculated on a per-object basis, and thus can be processed in a pipelined fashion.
3. Global Illumination: a much higher degree of realism can be achieved using global illumination, of course at the cost of speed. Global Illumination takes object interactions like reflections, transparency and shadows into consideration.
4. Image Processing: since the final result of the rendering process is an image, we will also discuss some aspects of 2D image processing, like e.g. filtering and compression.

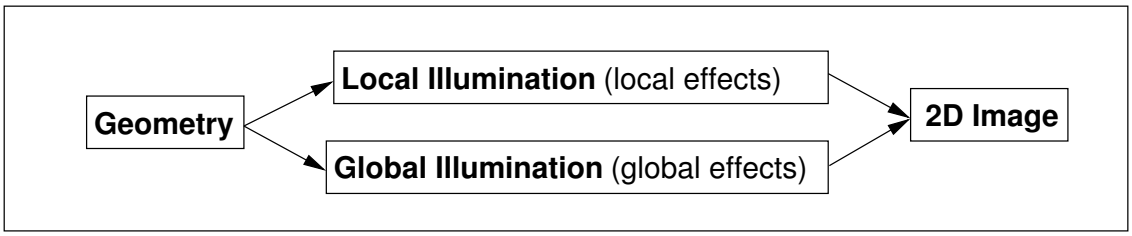


Figure 1.1: The different stages in Geometry Rendering.

However, since the efficient representation of data requires knowledge about how this data will be handled, we will discuss the local rendering pipeline first, and describe geometry, global illumination and image processing afterwards.

Of course we cannot cover all of these topics in depth, but after having attended this lecture you will be able to follow the more advanced courses in Computer Graphics. Some of the many topics which are of current research interest are the following:

- Geometric Modeling and Geometry Processing: finding efficient ways to represent and manipulate 3D objects mathematically.
- Rendering and Visualization: modeling processes or environments, and displaying them to the user in a comprehensive way.
- Image Processing: converting images and extracting information from them.
- Hardware: creating hardware to accelerate Computer Graphics applications.

1.2 Objects in 3D Space

Since we want to render three-dimensional geometric objects into two-dimensional images, we mostly work on the vector spaces $\mathbb{R}^3$ and $\mathbb{R}^2$. However, since we use other vector spaces as well, we briefly repeat some basic linear algebra facts in appendix A. In the following we will have a first look at how to represent the different objects we encounter in computer graphics.

The basic entity is the zero-dimensional *point* $p = (x, y, z)^T$. It is essentially a vector originating from the origin $(0, 0, 0)^T$, thus referring to a unique position in 3D space.

More-dimensional objects consist of an infinite set of points. There are generally two ways to specify these sets in a closed form:

- In the so-called *parametric form*, the object is defined as the *range* of a function, i.e. we specify a function $f : D \to \mathbb{R}^3$ and all points $f(x)$ with $x \in D$ belong to the object.
- One can also define the object to be the *kernel* of a function (the so-called *implicit form*), i.e. we specify a function $F : \mathbb{R}^3 \to \mathbb{R}$ and all points x with $F(x) = 0$ belong to the object. A simple implicit form is the distance to the object, i.e. $F_O(x) := dist(O, x)$.

Obviously, using the explicit form it is very easy to find all points on the object, but it requires some calculations to find out whether a given point x lies on the object. The implicit form yields the opposite: it is hard to find the points on the object, but trivial to check whether a given point lies on the object.

A *line* is one-dimensional. The most common representations are the following ones:

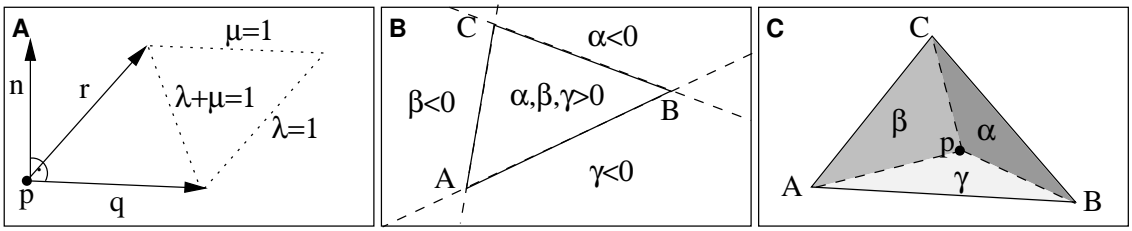


Figure 1.2: (a) planes, (b)(c) barycentric coordinates.

- Parametric: we can define a line explicitly by some base point p and the line's direction q:

$$L_{p,q} : \mathbb{R} \to \mathbb{R}^3, \quad \lambda \mapsto p + \lambda q$$

- Implicit (for lines in $\mathbb{R}^2$): we can give a condition that holds for each point x on the line. Let $L_{p,q}(\lambda) = p + \lambda q$ be the line in question, and n be a normal on q (i.e. $n \perp q$). Then we get for some point $x \in \mathbb{R}^2$ on $L_{p,q}$:

$$x = p + \lambda q \iff n^T x = n^T p + \lambda n^T q = n^T p \iff n^T x - n^T p = 0$$

$$L_{p,q} : \left\{ x \in \mathbb{R}^2 : n^T x - n^T p = 0 \right\}$$

  The form $n^T x - n^T p = 0$ is called *Normal Form*, and yields an implicit representation of the line. Analogously, we can specify a $(d-1)$-dimensional hyper-plane in $\mathbb{R}^d$ by its normal vector $n \in \mathbb{R}^d$, e.g. a plane in 3D.

- Implicit (for lines in $\mathbb{R}^3$): we can specify a line in 3D to be the intersection of two planes. Given the normals n_1 and n_2 of these planes, we get

$$L_{p,q} : \left\{ x \in \mathbb{R}^3 : n_1^T (x - p) = n_2^T (x - p) = 0 \right\}$$

Planes can be represented in a similar fashion; we can specify an origin p and two linearly independent vectors q and r, which span the plane (cf. Fig. 1.2a). Then, we get the following parametric/implicit representations:

$$P_{p,q,r} : \mathbb{R}^2 \to \mathbb{R}^3, \quad (\lambda, \mu) \mapsto p + \lambda q + \mu r,$$
$$P_{p,q,r} : \left\{ x \,|\, n^T x - n^T p = 0 \right\}, \quad p \in \mathbb{R}^3,\ n = q \times r.$$

Sometimes we want to restrict infinite lines or planes to certain finite subsets. These can be specified as follows:

- A *line segment* can be specified by constraining the parameter. Scaling the direction vector q properly, we can always specify a line segment to be

$$L_{p,q} \ : \ \{ p + \lambda q \mid \lambda \in [0, 1] \} .$$

- A *parallelogram* can be obtained in a similar way, constraining the two vectors q and r spanning the plane (cf. Fig. 1.2a):

$$P_{p,q,r} \ : \ \{ p + \lambda q + \mu r \mid \lambda, \mu \in [0, 1] \} .$$

- A *triangle* is essentially one half of a parallelogram. This can be achieved by adding another constraint (cf. Fig. 1.2a):

$$\triangle_{p,q,r} \ : \ \{ p + \lambda q + \mu r \mid \lambda, \mu \in [0, 1],\ \lambda + \mu \leq 1 \} .$$

Sometimes it is more convenient to specify a triangle by its three vertices A, B, C. With the above formula we obtain

$$\triangle_{A,B,C} : \{A + \lambda(B - A) + \mu(C - A) \mid \lambda + \mu \leq 1\}.$$

This can be rewritten to the following form

$$\triangle_{A,B,C} : \{\alpha A + \beta B + \gamma C \mid \alpha + \beta + \gamma = 1,\ \alpha, \beta, \gamma \geq 0\}$$

with $\alpha = 1 - \lambda - \mu$, $\beta = \lambda$, $\gamma = \mu$. Any affine combination $\alpha A + \beta B + \gamma C$ of A, B, C (i.e. $\alpha + \beta + \gamma = 1$) specifies a unique point p on the plane spanned by A, B, C; therefore (α, β, γ) are called the barycentric coordinates of p (with respect to A, B, C). All convex combinations yield points within the triangle. The different sectors outside the triangle are characterized by the signs of α, β and γ (cf. Fig. 1.2b).

More physically spoken, the barycentric coordinates (α, β, γ) of a point p w.r.t. a triangle specify the ratio of forces that must pull a mass point towards A, B and C, respectively, for its barycenter to be p. This implies that

$$\frac{\text{area}(\triangle(BCp))}{\alpha} = \frac{\text{area}(\triangle(ACp))}{\beta} = \frac{\text{area}(\triangle(ABp))}{\gamma},$$

as illustrated in Fig. 1.2c.

Barycentric coordinates in $\mathbb{R}^3$ are also defined for lines and n-gons with $n \geq 4$ in a similar fashion. Note, however, that for $n \geq 4$ the barycentric coordinates are not unique anymore.

Part I

Local Illumination

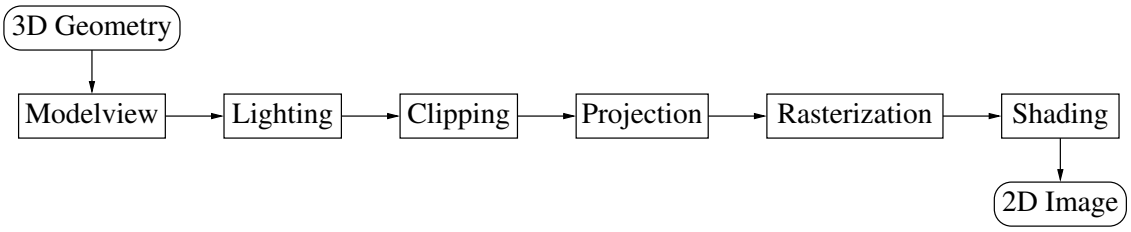


Figure 1.3: The Rendering Pipeline.

This part will be all about rendering 3D geometry into an image using *local illumination*. This means, we neglect all lighting effects on objects which are caused by the existence of other objects in the scene; such effects are shadows and reflections, for example. Another thing we cannot do without knowing all objects in the scene is determining which parts of the objects are actually visible. This can, for example, be solved by traversing all objects back to front. Other techniques will be presented in chapter 12.

The big advantage of local illumination is that we do not need to care about object interdependencies: we can do the rendering independently for each object. This enables us to process the geometry data in a pipelined (and even parallel) fashion: we can split the rendering process into several pipeline stages, and feed the geometry data into that pipeline object by object.

The speed-up we achieve by this simplification is so large that the reduced realism is often traded against it; for n objects, local illumination has a complexity of $\mathcal{O}(n)$, whereas global illumination requires at least $\mathcal{O}(n \log(n))$ (for the so-called *ray tracing*, cf. Sec. 14).

We can even go a step further: normally objects are composed from primitives. For reasons of efficiency, the primitives mainly used are points, lines, and planar polygons (triangles in most cases). In order to display curved surfaces we approximate them by polygons; spending enough polygons will yield a decent approximation. Therefore, we can feed these primitives into the pipeline, improving speed even further.

The different stages of this *rendering pipeline* (cf. Fig. 1.3) are listed below, and discussed in the following chapters:

1. *Transformations*: When rendering 3D geometry to a 2D image, including e.g. translations, rotations and projective mappings, several transformations are applied to an object until it appears on the screen. These transformations will be described in the next chapter and can be divided into the following categories:
   - *Model-to-World Transformations*: Usually a scene consists of many objects, each of which may be specified in its own local coordinate system. To compose the scene, we need to transform these coordinate systems into global world coordinates. This involves moving, resizing and re-orienting the primitives of the object.
   - *Viewing Transformations*: Since we are looking at the scene from some given camera position, we need to transform the global coordinate system into the local coordinate system of the camera. Model-to-World and Viewing Transformations together are generally referred to as *Modelview Transformations*.
   - *Projective Transformations*: Finally we use a projective mapping in order to display our three-dimensional objects on a two-dimensional image plane.
2. *Lighting*: In this step, we assign a color to each vertex in the scene. These lighting computations will use the so-called *Phong lighting model*. Note that all of these calculations are local, neglecting the existence of other objects in the scene.
3. *Clipping*: Since the displayable image is finite, it might happen that objects are only partly visible. Clipping refers to cutting off the invisible parts of an object, if necessary.

4. *Rasterization* refers to calculating the actual pixel positions for the vertices, lines and polygons on-screen and deciding which pixels to turn on or off.

5. *Shading*: We will calculate colors and brightnesses for the vertices only. Hence, we have to interpolate these values for the interior of a line or polygon. This process is called shading.

6. *Texturing*: Sometimes we might want to put an image (a *texture*) onto an object instead of just shading it; by doing so, we can greatly increase the visual detail.

Since the primitives we are sending down the rendering pipeline are points, lines and polygons, we will discuss the different subtasks of the rendering in this order: for point rendering, e.g., clipping gets trivial and we do not require shading or texturing. Rendering lines or polygons requires some more complicated methods, but can rely on some facts we already derived for points.

More detailed information on this topic can be found e.g. in the text books of Foley et al. [Foley et al., 2001] and Watt [Watt, 1999].

Chapter 2

Point Rendering Pipeline

The rendering pipeline for points is much simpler than the full rendering pipeline for polygons, since points do not have any spatial extent. After determining the position of the projected point on the image plane (modelview and projection) and computing its color (lighting) we can just set the resulting pixel's color. Therefore the clipping, rasterization and shading steps can be omitted (cf. Fig. 2.1).

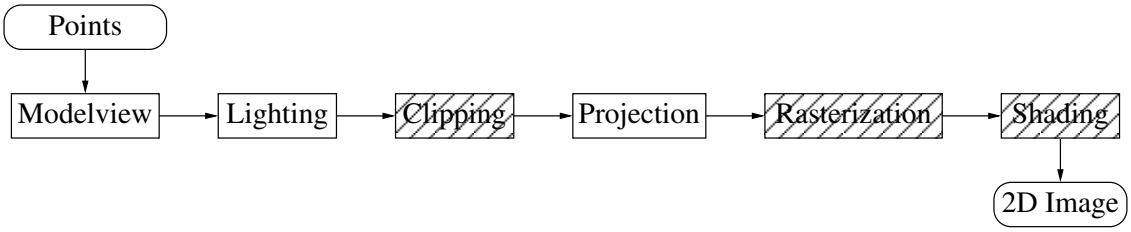


Figure 2.1: The Point Rendering Pipeline.

2.1 3D Transformations

In the following sections we will describe two kinds of 3D transformations: the so-called *modelview* transformations and the *projective* transformations.

The scene is usually composed of many objects. For matters of reuse and simplicity, these objects may be specified w.r.t. their own local coordinate system (*model coordinates*). For example, it is much more convenient to specify the polygon-model of a sphere w.r.t. its center than w.r.t. to its position in the scene. In addition, the scene might be structured in a hierarchy of local coordinate systems, e.g. for a solar system (sun, planets, moons) or for an arm of a robot (shoulder, upper arm, lower arm, hand). This means that we first have to map these local model-coordinates to global world coordinates using the *model-to-world transformation*. To view the scene from the desired position we then need a map from world coordinates into the coordinate system of our camera/observer (*viewing transformation*). The model-to-world transformation and the viewing transformation are usually combined in the so-called *modelview transformation*. We will see in sections 2.1.1 and 2.1.2 that these transformations can be represented by linear and affine maps.

On the other hand, we need some way to map three-dimensional geometry onto a two-dimensional image plane. This is done by *projective transformations* and will be discussed in section 2.1.4. We will see that there are several ways to perform this projection, depending on whether we aim at a realistically looking image or at a more technical drawing type, like e.g. those used in architectural applications.

In section 2.1.6 we will introduce a camera model that gives a very intuitive way of specifying the part of 3D space that is to be displayed on the screen.

By defining a set of camera parameters this is simply the part of the scene a camera would see from the specified position.

2.1.1 Linear Maps

The transformations we will describe in the following sections map 3D points to 3D points. As already mentioned in section 1.2 and in appendix A, points in 3-space are just elements of the three-dimensional Euclidean vector space $\mathbb{R}^3$.

As such, a natural type of transformations to consider are *linear maps*, since they preserve linearity. A map $L : \mathbb{R}^3 \to \mathbb{R}^3$ is called *linear*, if the following conditions are fulfilled:

1. Additivity: $L(p+q) = L(p) + L(q)\ ,\ \forall p, q \in \mathbb{R}^3$
2. Homogeneity: $L(\lambda p) = \lambda L(p)\ ,\ \forall \lambda \in \mathbb{R}, p \in \mathbb{R}^3$

This implies that linear maps leave the origin unaltered.

Since any vector p can be expressed as a linear combination of basis vectors (cf. appendix A), a linear map is uniquely determined by the images of the three standard basis vectors e_i and hence can be written as a matrix multiplication:

$$\begin{aligned} L(p) &= L(p_x e_1 + p_y e_2 + p_z e_3) \\ &= p_x L(e_1) + p_y L(e_2) + p_z L(e_3) \\ &= [L(e_1), L(e_2), L(e_3)] \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} =: M \cdot p \end{aligned}$$

Here, $M := [L(e_1), L(e_2), L(e_3)]$ is the matrix with $L(e_i)$ in its columns. If M is regular (i.e. $rank(M) = 3$), then the three transformed vectors $L(e_1)$, $L(e_2)$, $L(e_3)$ form another basis of $\mathbb{R}^3$. Therefore, such a linear map can also be regarded as a basis change.

In general, we do not want to transform a point by one map only, but instead define a complicated transformation to be the composition of several simpler maps. Fortunately, the composition of several linear maps is very easy. If we want to find the linear map L that combines the effect of applying the linear maps $L_1, \ldots, L_n$ one after the other, we just have to multiply the corresponding matrices $M_1, \ldots, M_n$:

$$\begin{aligned} L(p) &= (L_n \circ L_{n-1} \circ \ldots \circ L_2 \circ L_1)(p) \\ &= M_n \cdot M_{n-1} \cdot \ldots \cdot M_2 \cdot M_1 \cdot p \end{aligned}$$

For a given basis $\mathcal{B}$ consisting of orthonormal vectors $\{b_1, b_2, b_3\}$ (i.e. $\langle b_i, b_j \rangle = \delta_{ij}$) the matrix

$$_{\mathcal{E}}M_{\mathcal{B}} := [b_1, b_2, b_3]$$

can be regarded as a basis change from $\mathcal{B}$ to the standard basis $\mathcal{E}$. Any vector $p_{\mathcal{B}}$ w.r.t. to the basis $\mathcal{B}$ will be transformed into a vector $p_{\mathcal{E}} =_{\mathcal{E}} M_{\mathcal{B}} \cdot p_{\mathcal{B}}$, representing the same position, but now w.r.t. to the standard basis $\mathcal{E}$ (e.g. $(1,0,0)^T$ maps to b_1).

The inverse map $_{\mathcal{B}}M_{\mathcal{E}}$ from $\mathcal{E}$ to $\mathcal{B}$ is just the inverse of the respective matrix. Since $\mathcal{B}$ is assumed to be an orthonormal basis, the matrix $_{\mathcal{E}}M_{\mathcal{B}}$ is orthogonal and its inversion simplifies to a transposition:

$$_{\mathcal{B}}M_{\mathcal{E}} = (_{\mathcal{E}}M_{\mathcal{B}})^{-1} = (_{\mathcal{E}}M_{\mathcal{B}})^T$$

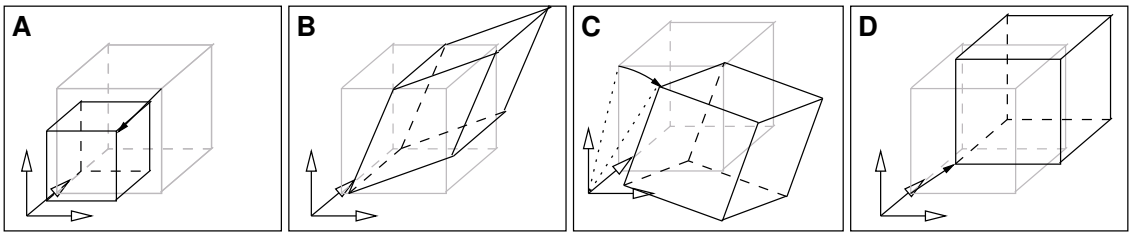


Figure 2.2: Model Transformations: (a) scaling, (b) xy-shearing, (c) rotation, (d) translation.

In the more general case, we want to map a vector from some orthonormal basis $\mathcal{B} = \{b_1, b_2, b_3\}$ to another orthonormal basis $\mathcal{Q} = \{q_1, q_2, q_3\}$. We decompose this transformation into two simpler ones: First we go from the basis $\mathcal{B}$ back to the standard basis $\mathcal{E}$. Then we map from $\mathcal{E}$ to the target basis $\mathcal{Q}$:

$$_{\mathcal{Q}}M_{\mathcal{B}} :=_{\mathcal{Q}} M_{\mathcal{E}} \cdot_{\mathcal{E}} M_{\mathcal{B}} = [q_1, q_2, q_3]^T [b_1, b_2, b_3]$$

Now that we know about linear maps, let us see what kind of 3D transformations we can represent with them. We will discuss the three main transformations represented by linear maps (cf. Fig. 2.2): *scalings* (resizing an object), *shearings* (tilting an object along an axis) and *rotations* (turning an object around an axis).

Scaling

Scaling means changing the size of the object by multiplying each of its coordinates by a certain factor. Normally, scaling is performed by the same factor for all coordinate axes, but of course one can specify different scaling factors for each of them.

Since scaling is a linear map, the origin remains unchanged and therefore is the fixed point or scaling center of this transformation. The matrix for a scaling along the coordinate axes with factors α, β, and γ is given by

$$S(\alpha, \beta, \gamma) := \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \beta & 0 \\ 0 & 0 & \gamma \end{pmatrix}$$

and we also define $S(\alpha) := S(\alpha, \alpha, \alpha)$ for uniform scaling (Fig. 2.2a). Obviously, setting the factor to one yields the identity, a factor smaller than one results in shrinking, and a factor larger than one results in stretching. Scaling an object preserves parallelism of the object's lines; however, lengths are obviously not preserved for $\alpha \neq 1$. Angles are preserved iff the applied scaling is uniform, i.e. $\alpha = \beta = \gamma$.

In order to scale an object w.r.t. a given axis or direction, we have to rotate this axis to one of the coordinate axes first. Rotations will be described in a minute.

Shearing

Shearing an object means tilting it along one (or several) axes. For example, shearing in x- and y-direction yields the following matrix (cf. Fig. 2.2b):

$$Sh_{xy}(\alpha, \beta) = \begin{pmatrix} 1 & 0 & \alpha \\ 0 & 1 & \beta \\ 0 & 0 & 1 \end{pmatrix}$$

Shearing does not preserve lengths and angles, but it does preserve parallelism of lines.

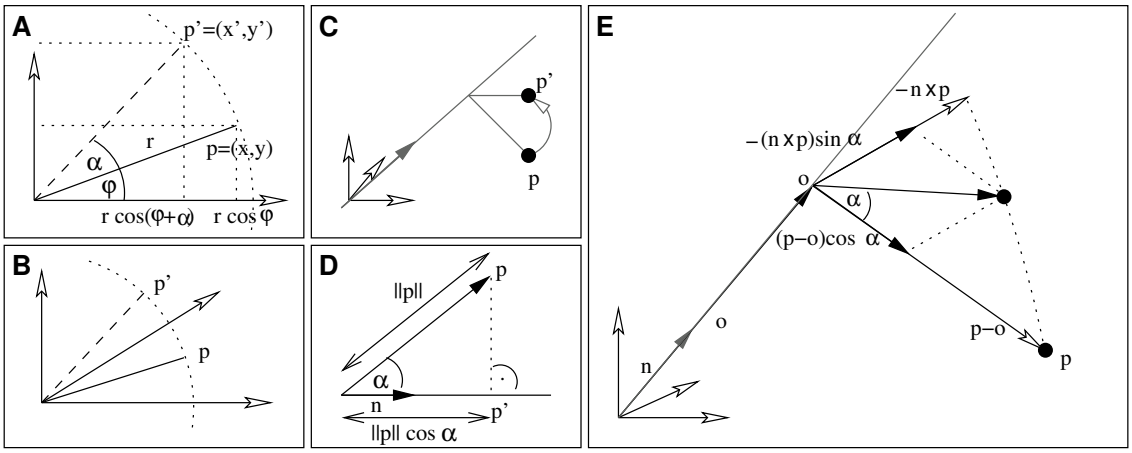


Figure 2.3: Rotation: (a) 2D, (b) in xy-plane, (c)(e) around arbitrary axis (d) projection.

Rotation

Let us consider the 2D case first: rotating a point $p = (x, y)^T$ by an angle of α around the origin $(0, 0)^T$ yields a new point $p' = (x', y')^T$, which can be obtained as follows. Letting $r = \|p\|$, we can write both points in polar coordinates $(r, \phi)^T$ and $(r, \phi + \alpha)^T$ (cf. Fig. 2.3a).

Using basic theorems of trigonometry we then get:

$$\begin{aligned} x' &= r \cdot \cos(\varphi + \alpha) \\ &= r \cdot \cos\varphi\cos\alpha - r \cdot \sin\varphi\sin\alpha \\ &= x \cdot \cos\alpha - y \cdot \sin\alpha \\ y' &= r \cdot \sin(\varphi + \alpha) \\ &= r \cdot \sin\varphi\cos\alpha + r\cos\varphi\sin\alpha \\ &= y \cdot \cos\alpha + x \cdot \sin\alpha \end{aligned}$$

If we add another dimension and rotate around the z-axis (cf. Fig. 2.3b), the z-coordinate remains unchanged, and the other coordinates are transformed according to the 2D case. This yields the rotation matrix

$$R_z(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Rotating around the other axes yields similar matrices:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{pmatrix}, \quad R_y(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha \\ 0 & 1 & 0 \\ -\sin\alpha & 0 & \cos\alpha \end{pmatrix}$$

Rotating a point p around an arbitrary axis $n = (n_x, n_y, n_z)^T$ with $\|n\| = 1$ requires a bit more computation, since we need to set up a proper local coordinate system. The origin o will be the projection of p onto n, the axes naturally get $p - o$, $-(n \times p)$, and n (cf. Fig. 2.3c).

The orthogonal projection of p onto n can be calculated by $o = n \cdot n^T \cdot p$ (cf. Fig. 2.3d). Furthermore, since the cross product is a linear operator, it can be written as a matrix:

$$-(n \times p) = -X(n) \cdot p =: \begin{pmatrix} 0 & n_z & -n_y \\ -n_z & 0 & n_x \\ n_y & -n_x & 0 \end{pmatrix} \cdot p$$

Now, the position vector of p' can be obtained according to Fig. 2.3e:

$$\begin{aligned} p' &= o + \cos\alpha \cdot (p - o) - \sin\alpha \cdot (n \times p) \\ &= nn^T p + \cos\alpha \cdot (p - nn^T p) - \sin\alpha \cdot X(n) \cdot p \\ &= nn^T p + \cos\alpha \cdot p - \cos\alpha \cdot nn^T p - \sin\alpha \cdot X(n) \cdot p \\ &= \left[\cos\alpha \cdot I + (1 - \cos\alpha) \cdot nn^T - \sin\alpha \cdot X(n)\right] \cdot p \end{aligned}$$

Thus the matrix for rotating around n and α gets:

$$R(n, \alpha) = \cos\alpha \cdot I + (1 - \cos\alpha) \cdot nn^T - \sin\alpha \cdot X(n)$$

Rotations do not change the object's shape: parallel lines remain parallel, and lengths and angles are preserved (since rotations are orthogonal matrices). However, the orientation and position of the object changes.

2.1.2 Affine Maps

In the last section we have seen that linear maps always leave the origin unchanged. As a consequence, it is not possible to write a translation by a certain offset $(t_x, t_y, t_z)^T$ as a linear map. Translations are so called *affine maps*, which are essentially linear maps followed by a translation:

$$A(p) = L(p) + t, \ p, t \in \mathbb{R}^3$$

Unfortunately, we cannot write an affine map as a matrix multiplication. Since the concatenation of several transformations into one accumulated matrix is very convenient and extremely important for the efficient implementation of transformations, we will introduce *extended coordinates* that will provide a way of representing affine maps by matrices.

Let us first have a closer look at points and vectors. While both elements are often considered to be equivalent, i.e. elements of the vector space $\mathbb{R}^3$, they have in fact different geometric meanings. Points, on the one hand, refer to positions in 3-space; vectors, on the other hand, are offsets or directions between two points. The connection is, that we can consider points to be vectors originating from the origin $(0, 0, 0)^T$. Although we can arbitrarily combine vectors and points, not all of these operations are geometrically meaningful:

- Adding a vector to a point yields a point: this means moving the point by an offset.
- Adding a vector to another vector yields a vector; obviously, moving a point by an offset v, and then moving it by an offset w is the same as moving it by $(v + w)$ at once.
- Subtracting a point from another one yields the offset one must move the first point to get to the second one. Thus, the result of this operation is a vector.
- Adding two points is generally ambiguous: We get different results depending on the coordinate system we are using.

Extended coordinates therefore distinguish between points and vectors by adding a fourth component to the respective coordinates. This fourth coordinate is 1 for points and 0 for vectors. This might be interpreted as two three-dimensional hyper-planes in $\mathbb{R}^4$ that are fixed by the fourth component. Using extended coordinates, the result of a combination of points/vectors is geometrically meaningful if the last component of the result is either zero or one, i.e. it can be interpreted as a 3D vector or point, respectively.

While, in general, this is not true for linear combinations of points, there are important exceptions. Having a closer look at the linear combination $\sum_i \alpha_i p_i$ (with $p_i \in \mathbb{R}^4$ represented in extended coordinates) we see that the fourth component of the result is $\sum_i \alpha_i$. Therefore, this linear combination has a geometric meaning only if it is an affine combination, i.e. the $\sum_i \alpha_i = 1$.

There is another advantage of extended coordinates: affine maps in the linear space $\mathbb{R}^3$ can now be written as linear maps in the affine space $\mathbb{R}^4$:

$$A(p) = L(p) + t = \begin{pmatrix} L_{11} & L_{12} & L_{13} & t_1 \\ L_{21} & L_{22} & L_{23} & t_2 \\ L_{31} & L_{32} & L_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix}$$

Analogously, any linear map represented by a 3×3 matrix can be written as the 4×4 matrix of an affine map by setting the translational component to $(0, 0, 0)^T$. In this way all linear maps we derived in the last section can also be used in connection with extended coordinates. Using this notation we can again use matrix multiplication to compose several maps.

A translation by an offset $t = (t_x, t_y, t_z)^T$ can now be written as

$$T(t) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

In order to uniquely define a 4×4 affine matrix M, we now need the images of the four standard basis vectors $e_1, \ldots, e_4$. Analogously to linear maps, the images $M \cdot e_i$ will build the columns of M. Looking at the fourth component of the e_i's we understand their geometric meaning: the first three vectors e_1, e_2, e_3 have a zero in the fourth component, hence they represent vectors (the coordinate frame of the new basis). The vector e_4 represents the 3D point $(0, 0, 0)^T$, i.e. the origin. As a consequence, an affine basis change is specified by the directions of the axes and the position of the origin.

2.1.3 Concatenating Transformations

Since all transformations we discussed so far can be represented by matrices using extended coordinates we can simply concatenate them by multiplying the corresponding matrices. Thus, the transformation matrix M which successively applies the transformations $M_1, M_2, \ldots, M_n$ can be written as

$$M = M_n \cdot M_{n-1} \cdot \ldots \cdot M_2 \cdot M_1.$$

But since matrix multiplication is in general not commutative, the order of the transformation matters. This can easily be seen by considering a rotation R and a translation T. The transformation $T \cdot R$ would rotate the object (around the origin) and translate it afterwards. $R \cdot T$ would first move the object away from the origin; since the following rotation will still be around the origin, the results will differ.

Let us recapitulate what the modeling and viewing transformations do: they transform objects in local coordinate systems into the global coordinate system of the camera. We could get the same *relative* result, if we transformed the global coordinate system (i.e. the camera) into the local coordinate system of the object. In the first case we transform the object to fit into the scene, in the second case we move the camera relative to the object. Both versions are equivalent, but you have to keep in mind that the operations are inverse; translating an object by t is equivalent to moving the camera by $-t$ (cf. Fig. 2.4).

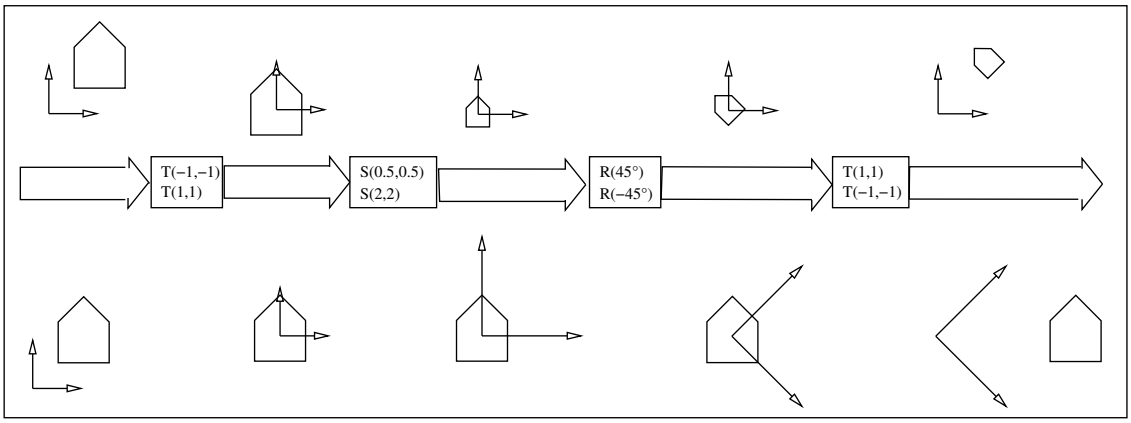


Figure 2.4: Local vs. Global Coordinate System.

2.1.4 Projective Maps

In the last section we discussed how to transform points or objects in 3-space. The transformations we derived could be represented as matrix multiplications using extended coordinates. Since the final image we want to render has only two dimensions we have to apply some projective map. Projections of 3D objects onto a 2D image plane will be the topic of this section.

There are two main types of projections: *perspective projections* model the human way of viewing, including perspective foreshortening of objects far away. Because of the fact that all viewing rays start from the viewer's eye, this projection is also referred to as central projection. The other type are *parallel projections*, widely used in technical applications. Here, all the viewing rays are parallel, causing the objects to have original size on the image plane, regardless of their distance from the viewer.

In the following sections we will take a closer look at these two types of projections. But first we will face the same problem we had when dealing with affine maps: we could not represent them by matrices until we introduced extended coordinates.

Homogeneous Coordinates

The problem of representing projections using matrices can already be discussed considering the *standard projection*. In this case the center of projection (i.e. the camera/viewer) is located at the origin, the viewing direction is the negative z-axis $(0, 0, -1)^T$ and the projection plane is defined by $z = -1$. The map performing the corresponding projection is

$$(x, y, z)^T \mapsto \left(\frac{x}{-z}, \frac{y}{-z}, -1\right)^T.$$

Since this projection includes divisions and since matrix multiplications are only able to represent additions and scalar multiplications, we cannot (yet) use matrices to represent projective maps. We will therefore introduce *homogeneous coordinates* that are a further extension of extended coordinates. This kind of representation then enables us to specify projections/divisions in matrix notation.

In the homogeneous coordinate system we define a point $(x, y, z)^T \in \mathbb{R}^3$ to be equivalent to the line $(wx, wy, wz, w)^T \in \mathbb{R}^4$ for any $w \neq 0$. Any two points $p = (x_1, y_1, z_1, w_1)^T$ and $q = (x_2, y_2, z_2, w_2)^T$ represent the same 3D point, if $p = \alpha q$ for some $\alpha \in \mathbb{R} \setminus \{0\}$.

For each 3D point we define a special representative on the corresponding line in $\mathbb{R}^4$ having the fourth component equal to 1, and we call that representation *de-homogenized*. We get this

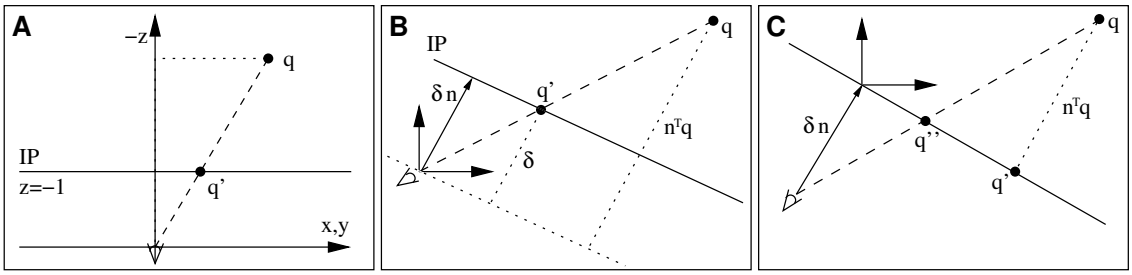


Figure 2.5: Perspective Projections: (a) standard projection, (b) arbitrary image plane, viewer at the origin, (c) arbitrary image plane at the origin, viewer at $-\delta n$.

de-homogenized representation by dividing the point by its fourth component. Analogously to extended coordinates, vectors will have the fourth component set to 0, and can be interpreted as points at infinity ($(x, y, z, 0) \cong \lim_{w\to 0}(x/w, y/w, z/w)^T$).

Using this representation we can multiply the fourth component by the dividend instead of dividing by it. All divisions during the transformations accumulate, and are executed once during de-homogenization.

$$P((x, y, z)^T) = M \cdot (x, y, z, 1)^T = (x', y', z', w') \cong \left(\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'}, 1\right).$$

Homogeneous coordinates embed the 3D Euclidean space into the 4D projective space, thereby providing the possibility to represent linear, affine, and projective maps by 4×4 matrices. Again, concatenation of several of these maps can be done by simply multiplying the respective matrices in the right order.

In the following sections we will now derive the different projective maps and their corresponding projection matrices.

Perspective Projections

As already mentioned in the previous section, *perspective projections* model the way we look at a scene by causing distant objects to appear smaller (so-called *perspective foreshortening*). The last section introduced the *standard projection*. Here, the camera is located at the origin, the image plane is defined by $z = -1$, and the viewing direction is $(0, 0, -1)^T$ (cf. Fig. 2.5a).

The projection of an object point $(x, y, z)^T$ onto the image plane can be obtained by:

$$(x, y, z)^T \mapsto \left(\frac{x}{-z}, \frac{y}{-z}, -1\right)^T$$

Using homogeneous coordinates we get the following transformation matrix:

$$P_{\text{std}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

Multiplying a point by this matrix and performing the de-homogenization afterwards in fact has the desired effect of division:

$$P_{\text{std}} \cdot (x, y, z, 1)^T = (x, y, z, -z)^T \cong (\frac{x}{-z}, \frac{y}{-z}, -1, 1)^T \to (\frac{x}{-z}, \frac{y}{-z})^T.$$

We will continue with a little bit more complicated example: we again choose the origin to be the viewing position, but specify an arbitrary image plane by its normalized normal $n = (n_x, n_y, n_z)^T$ and the focal distance δ from the origin (cf. Fig. 2.5b). For a given point $q = (x, y, z)^T$ the projected point q' is

$$q' = \frac{\delta}{n^T q} \cdot q.$$

This corresponds to the following projection matrix:

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{n_x}{\delta} & \frac{n_y}{\delta} & \frac{n_z}{\delta} & 0 \end{pmatrix}.$$

In a last step we consider the case in which the image plane with normal n is centered at the origin, and the center of projection is located at $-\delta n$, δ again denoting the focal distance (cf. Fig. 2.5c). Using the notation of Fig 2.5c we first get q' by orthogonally projecting q onto the image plane: $q' = (I - nn^T)q$. The final projected point q'' is then just a scaled version of q': $q'' = q' \cdot \frac{\delta}{\delta + n^T q}$. The complete projection is

$$q \mapsto \left(q - nn^T q\right) / \left(\frac{n^T q}{\delta} + 1\right).$$

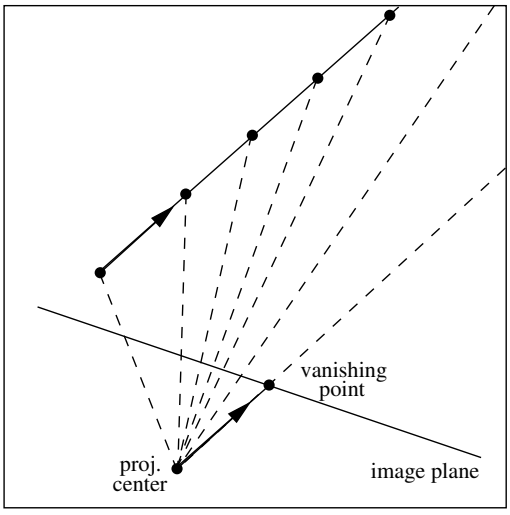
This projection can be represented by the following matrix, where the upper 3×3 block is $(I - nn^T)$ (yielding q') and the fourth row causes the respective scaling:

$$\begin{pmatrix} 1 - n_x^2 & -n_x n_y & -n_x n_z & 0 \\ -n_x n_y & 1 - n_y^2 & -n_y n_z & 0 \\ -n_x n_z & -n_y n_z & 1 - n_z^2 & 0 \\ \frac{n_x}{\delta} & \frac{n_y}{\delta} & \frac{n_z}{\delta} & 1 \end{pmatrix}.$$

For a *parallel projection* all viewing rays will be parallel, corresponding to a viewing position that is infinitely far away. From the last projection we can construct a parallel projection by increasing the focal distance δ to infinity. The result will be the orthogonal projection $(I - nn^T)$ onto the image plane in form of a 4×4 matrix.

Vanishing Points

An effect caused by perspective foreshortening is that parallel lines can meet at a certain distant point, compare e.g. a train's rails. The point in which these parallel lines of direction d meet is called the *vanishing point* corresponding to the direction d.



As we will see in a second, this point can be constructed by intersecting the line parallel to d and starting at the viewing position with the image plane (see figure). This implies that all parallel lines have the same vanishing point.

What is the mathematical reason for this effect? To see this, let $L(\lambda) = (x, y, z)^T + \lambda \cdot (d_x, d_y, d_z)^T$ be some line in object space. Using the standard projection P_{std}, the projection of the point $L(\lambda)$ is

$$P_{\text{std}}(L(\lambda)) = \begin{pmatrix} \frac{x+\lambda d_x}{z+\lambda d_z} \\ \frac{y+\lambda d_y}{z+\lambda d_z} \end{pmatrix}$$

What happens when we approach infinity, i.e. $\lambda \to \infty$? There are two cases:

- If $d_z = 0$ (which means that the line is parallel to the image plane), then

$$P_{\text{std}}(L(\lambda)) = \begin{pmatrix} \frac{x+\lambda d_x}{z} \\ \frac{y+\lambda d_y}{z} \end{pmatrix} \xrightarrow{\lambda \to \infty} \begin{pmatrix} \infty \\ \infty \end{pmatrix}$$

  The interpretation is that there is no vanishing point if the line is parallel to the image plane.

- If $d_z \neq 0$ (which means that the line is not parallel to the image plane), then

$$P_{\text{std}}(L(\lambda)) = \begin{pmatrix} \frac{x+\lambda d_x}{z+\lambda d_z} \\ \frac{y+\lambda d_y}{z+\lambda d_z} \end{pmatrix} = \begin{pmatrix} \frac{x}{z+\lambda d_z} \\ \frac{y}{z+\lambda d_z} \end{pmatrix} + \begin{pmatrix} \frac{d_x}{z/\lambda+d_z} \\ \frac{d_y}{z/\lambda+d_z} \end{pmatrix} \xrightarrow{\lambda \to \infty} \begin{pmatrix} \frac{d_x}{d_z} \\ \frac{d_y}{d_z} \end{pmatrix}$$

  The point we get is the vanishing point corresponding to $(d_x, d_y, d_z)^T$. This point is the same for parallel lines, since it does not depend on the starting point of the line $(x, y, z)^T$.

Projective transformations can now be categorized by the number of (finite) vanishing points of the coordinate axes, i.e. the number of coordinate axes that are not parallel to the image plane.

In the next subsections we will discuss how to *construct* (instead of calculate) perspective transformations for one-, two- and three-point perspectives. In all of these cases we draw an axis-aligned cube with edge length one; from this cube we can then start to construct other objects, like e.g. houses. The elements we will use for this construction are:

- The main point h is the orthogonal projection of the viewing position onto the image plane.
- The focal distance δ is the distance from the viewing position to the image plane.
- The vanishing points z_i of the coordinate axes e_i. We will also make use of the vanishing points d_i of the diagonals $(1, \pm 1, 0)^T$, $(0, 1, \pm 1)^T$ and $(1, 0, \pm 1)^T$.

One-Point Perspective In case of a one-point perspective two of the coordinate axes are parallel to the image plane. Our goal is to draw an axis-aligned cube of edge length 1. In order to construct this drawing we are given the focal distance δ and the main point h. A one-point perspective implies that the front face of the cube is parallel to the image plane. We also know that the vanishing point of the z-axis is the main point h. The only information we additionally need to know are the vanishing points of the diagonals on the side faces.

We already discussed that the vanishing point of a direction d can be constructed by intersecting a line parallel to d and starting at the camera with the image plane. Knowing that the diagonals have a 45° angle to the image plane and that the camera center has distance δ from the image plane, we can derive that the diagonal vanishing points have distance δ from the main point. So d_1 and d_3 are at position h shifted by δ into x- and y-direction, respectively.

We can now construct the unit cube: we draw the front face, and connect its vertices to the vanishing point z_3. Next we connect the vertices with the vanishing points d_1 and d_3 of the diagonals; the intersection of them with the lines towards z_3 give us the position of the back face (cf. Fig. 2.6a).

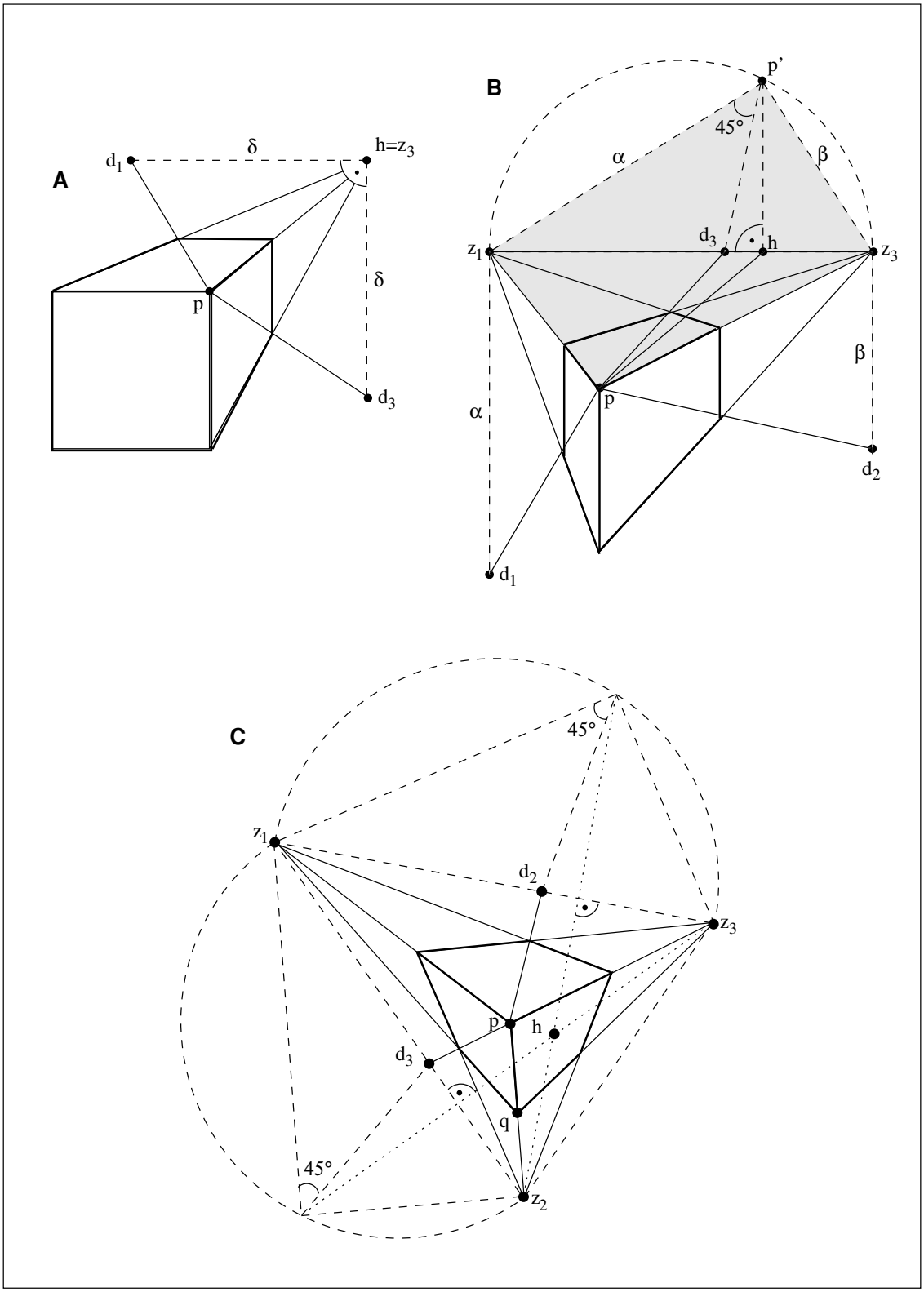


Figure 2.6: Constructing perspective drawings in the case of a (a) 1-point perspective, (b) 2-point perspective, (c) 3-point perspective.

Two-Point Perspective In case of a two-point perspective one edge of the cube is parallel to the image plane (let p be the upper point of this edge). In this setting we are given the main point h and the vanishing points z_1 and z_3. Note that these three points lie on the horizon line.

We now first note that the diagonal vanishing point d_3 will be on the horizon, too, and we can obtain its position as follows.

We rotate p around z_1z_3 into the image plane to get p'; the angle $\angle(z_1 - p, z_3 - p) = 90°$ will be preserved, i.e. $\angle(z_1 - p', z_3 - p')$ also has to be 90°. Therefore, all possible triangles z_1z_3p' can be constructed by a Thales circle. We also know that the line z_1z_3 has to be perpendicular to the line hp'. Therefore p' is the intersection of the Thales circle around z_1 and z_3 and the perpendicular line starting from h. The rotated diagonal must be the bisecting line of the angle at p', and intersecting this line with the horizon yields d_3. The other two diagonal vanishing points must be chosen such that $z_1d_1 \perp z_1z_3$ and $z_3d_2 \perp z_1z_3$ and that $\|p' - z_1\| = \|z_1 - d_1\|$ and $\|p' - z_3\| = \|z_3 - d_2\|$. Having constructed the positions of the three diagonal vanishing points we can now finish the drawing (cf. Fig. 2.6b).

Three-Point Perspective In the last case, none of the coordinate axes (cube edges) is parallel to the image plane. This time, we are given the vanishing points z_1, z_2 and z_3 of all the coordinate axes. Using similar arguments as in the two-point case we can derive that the main point is the intersection of the perpendicular bisectors of the triangle $\Delta(z_1, z_2, z_3)$.

Following the ideas used for the two-point perspective, we can again construct the rotations of the diagonal triangles into the image plane. This gives us all the diagonal vanishing points, and we can draw the cube (cf. Fig. 2.6c).

Parallel Projections

In contrast to a perspective projection (where all viewing rays meet at the projection center) the viewing rays in a *parallel projection* are, as the name indicates, parallel to each other. As a consequence there is no perspective foreshortening, i.e. objects keep their size regardless of their distance from the viewer. This property is very handy for technical applications, since it allows measuring objects' sizes in drawings. On the other hand, this of course implies a lack of realism, because it does not model the way we look at a scene.

Parallel projections can be seen as a special case of perspective projections, with the eye infinitely far away from the image plane. There are two kinds of parallel projections, *orthogonal* and *oblique projections*, which are categorized based on whether or not the viewing rays are orthogonal to the image plane.

Orthogonal Projections For *orthogonal projections* the viewing rays are orthogonal on the image plane. We will restrict ourselves to the case in which the image plane is orthogonal to one of the coordinate axes. This gives us three different orthogonal projections widely used in the area of architecture: the *top view* $y = 0$ ("*Grundriss*"), the *front view* $z = 0$ ("*Aufriss*"), and the *side view* $x = 0$ ("*Kreuzriss*") (cf. Fig. 2.7a).

Obviously, these transformations have quite simple projection matrices: the only thing we have to do is to set the corresponding entry in the vector to zero. In case of the top view we get for instance

$$P_{\text{top}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

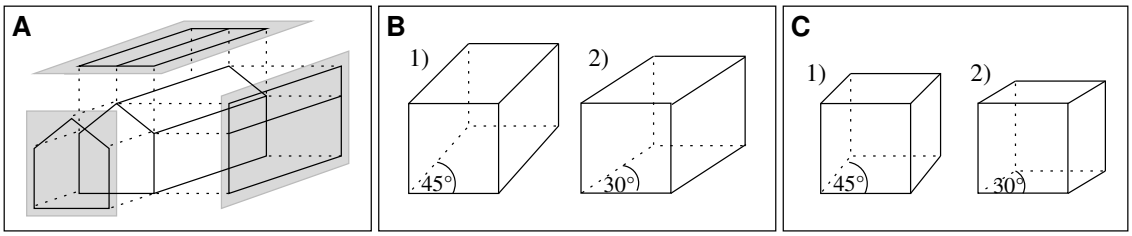


Figure 2.7: Parallel Projections: (a) orthogonal, (b) cavalier, (c) cabinet.

Oblique Projections Orthogonal projections have the problem that they do not give any information about the depth of objects. *Oblique transformations* give a better impression of depth by letting the viewing rays cross the image plane with an angle other than 90°; frequently used angles are 30° and 45°. There are two kinds of oblique projections, the *cavalier projection* and the *cabinet projection*, differing in a scaling factor for the depth coordinate.

In the *cavalier projection* (cf. Fig. 2.7b) the length of the lines that are not parallel to the image plane does not change. This fact provides a very easy way to measure the length of these lines, but it gives a very bad impression of depth. In the *cabinet projection* (cf. Fig. 2.7c), these lengths are halved; the lines' lengths are still easy to measure (just double their sizes), but this method gives a much better feeling of the objects' depth. Note that in both cases the scaling factors are constant and independent from the depth, in contrast to perspective projections.

The matrix for a 45° cabinet projection is

$$P_{\text{cab}} = \begin{pmatrix} 1 & 0 & \frac{1}{2}\cos 45° & 0 \\ 0 & 1 & \frac{1}{2}\cos 45° & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

2.1.5 2D Coordinates

Up to now we discussed how to map 3D points onto an image plane. The resulting coordinates still had three or four (homogeneous coordinates) components. In order to get rid of the unnecessary depth component, we have to construct a local coordinate system of the image plane and represent the projected point w.r.t. to this coordinate system.

One point m and two perpendicular unit vectors u and v on the image plane build this coordinate system:

$$IP(x, y) = m + xu + yv.$$

Given any point p on the image plane, we can compute its 2D coordinates by

$$x = u^T(p - m), \;\; y = v^T(p - m).$$

This basis change can be written as a matrix, mapping the result of the projection step to homogeneous screen coordinates

$$\begin{pmatrix} wx \\ wy \\ w \end{pmatrix} = \begin{pmatrix} u_x & u_y & u_z & -u^T m \\ v_x & v_y & v_z & -v^T m \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}.$$

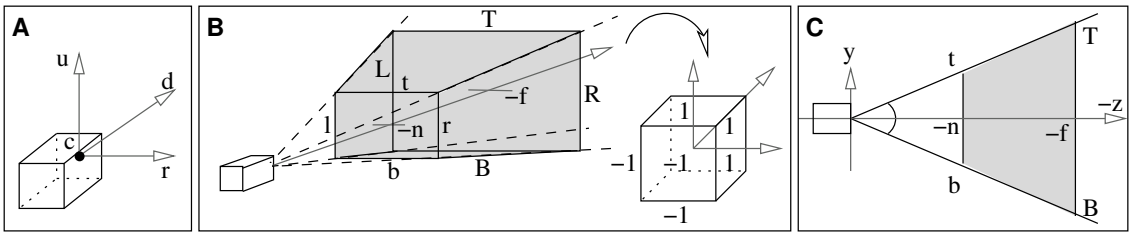


Figure 2.8: Camera Model: (a) local coordinate system, (b) frustrum transformation, (c) 2D.

2.1.6 Camera Model

Specifying the projection used to map a given 3D scene onto a 2D image plane can be quite tedious, since we have to provide camera center, focal distance, normal vector of the image plane, and finally a proper 2D coordinate system. The situation becomes even more complicated when we want to restrict the visible space, e.g. by using some clipping planes.

It is much more intuitive to consider a camera. Then all the required values can be given as extrinsic and intrinsic parameters of the camera: position and orientation, focal distance, opening angles. We will also use a minimum and maximum distance to further restrict the visible area; this corresponds to a near- and a far plane orthogonal to the viewing axis of the camera.

The transformation corresponding to a given camera setting can be split up into three parts:

1. The *Look-At transformation* transforms the global coordinate system into the local coordinate system of the camera. In this coordinate system the camera is located in the origin and looks into the negative z-direction. Therefore this essentially builds the viewing transformation.

2. The *Frustrum transformation* projectively warps the viewing volume of the camera (i.e. a frustrum determined by opening angles, near- and far plane) to the unit cube $[-1,1]^3$. Because of this projective warping the perspective projection simplifies to a parallel projection after the frustrum transformation.

3. The *Viewport transformation* performs the remaining parallel projection and maps the resulting 2D coordinates from $[-1,1]^2$ to the size of the viewport, i.e. the window pixel coordinates.

Look-At Transformation

A camera's orientation in space can be specified by a position, a viewing direction and a direction pointing "up" in the camera coordinate system. Without the up-direction the camera could still be rotated around the viewing direction (cf. Fig. 2.8a).

Given a camera position and its orientation, the look-at transformation performs a basis change such that the scene is transformed into the setup of the standard projection. Hence, the camera is shifted into the origin and oriented such that it looks down the negative z-axis and that the y-axis is the up-direction.

First of all we have to construct a local coordinate system given the camera settings. The origin of this coordinate system will be the camera position c. Its three axes are the viewing direction d, the up-vector u and the right-vector r. Given the vectors d and u, the right vector is perpendicular to them, i.e. $r = d \times u$. Since u may not be perpendicular to d (it only must not be parallel to d) we use a second cross product to make it perpendicular to both d and r: $u = r \times d$. After normalizing these three vectors, c, d, u and r now specify an orthogonal affine coordinate system.

As shown in sections 2.1.1 and 2.1.2 the matrix corresponding to the map of the standard basis $\mathcal{E}$ into the camera coordinate system $\mathcal{C}$ consists of the images of the four affine basis vectors e_1, e_2, e_3, e_4. This yields the matrix

$$ {}_{\mathcal{E}}M_{\mathcal{C}} = \begin{pmatrix} r_x & u_x & -d_x & c_x \\ r_y & u_y & -d_y & c_y \\ r_z & u_z & -d_z & c_z \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

The matrix we are looking for is the inverse of this matrix (this is *not* the transposed, since the vectors are orthogonal, but not orthonormal):

$$ M_{\text{LookAt}} = {}_{\mathcal{C}}M_{\mathcal{E}} = \begin{pmatrix} r_x & r_y & r_z & -r^T c \\ u_x & u_y & u_z & -u^T c \\ -d_x & -d_y & -d_z & d^T c \\ 0 & 0 & 0 & 1 \end{pmatrix} $$

Frustrum Transformation

What is the part of the scene a camera located at the origin and looking along the negative z-axis can see? Since the camera's image is rectangular, we obtain a pyramid, with the apex in the origin, and opening along the negative z-axis. In order to prescribe a minimum and maximum distance for visible objects, we add a *near-* and *far plane* orthogonal to the z-axis at $z = -n$ and $z = -f$, respectively: only objects in between these planes are visible for the camera. The resulting viewing volume is a truncated pyramid, i.e. a frustrum (cf. Fig. 2.8b).

This frustrum models the intrinsic camera parameters, i.e. focal distance, opening angles, and valid distance. The opening angles are specified by defining a rectangle on the near plane. Let its top coordinate be t, its bottom-coordinate b, its left-coordinate l and its right-coordinate r (all coordinates are scalars).

The frustrum transformation now projectively warps this frustrum onto the cube $[-1, 1]^3$ (cf. Fig. 2.8b); the matrix corresponding to that transformation is

$$ M_{\text{frustrum}} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} $$

After this transformation, the near plane is the front-plane $z = -1$ of the cube, the far plane is its backplane $z = 1$. This transformation is projective: although the far plane is larger than the near plane, both planes have the same size in the cube, thus objects near the far plane are shrunk (resulting in perspective foreshortening). In fact, performing a perspective projection is equivalent to applying a frustrum transformation followed by a parallel projection.

Obviously, there are many ways to parameterize the frustrum matrix; the one above is specified in terms of the near- and far plane and the limiting coordinates b, t, l, and r. However, it is also quite common to use other parameters:

- Instead of giving the t/b/r/l-coordinates, it is also possible to specify the *opening angle* of the camera. With a vertical opening angle of φ (also called "fovy": *field of view* in y-direction) and a horizontal opening angle of θ ("fovx"), we get the following connection:

$$ t = n \cdot \tan\frac{\varphi}{2} \qquad b = -n \cdot \tan\frac{\varphi}{2} $$
$$ r = n \cdot \tan\frac{\theta}{2} \qquad l = -n \cdot \tan\frac{\theta}{2}. $$

Note that this always results in a symmetric frustrum, i.e. $r = -l$ and $t = -b$. Although this is the most intuitive setting, it may be too restrictive for certain applications.

- Instead of the coordinates on the near plane, we could also specify those of the far plane; their values can be obtained by

$$T = t \cdot \frac{f}{n}, \;\; B = b \cdot \frac{f}{n}, \;\; L = l \cdot \frac{f}{n}, \;\; R = r \cdot \frac{f}{n}.$$

- Since the near- and far plane are rectangular, we could also specify the height by an opening angle and the width in terms of an *aspect ratio* ($a := w/h$). Using that, we get

$$t = n \cdot \tan \frac{\varphi}{2}, \;\; b = -t, \;\; r = a \cdot t, \;\; l = -r.$$

The Viewport Transformation

The last thing to be done is mapping the cube onto the viewing plane, and then resizing that plane to match the size of the *viewport* (this is the part of the screen in which the image is to be displayed).

The first step is quite easy: since we now only need to perform a parallel projection, we are done by leaving away the z-coordinate (normally we just neglect that coordinate, or keep it e.g. for later occlusion tests). After projecting the cube's interior $[-1, 1]^3$ onto its front face the image has coordinates $[-1, 1]^2$.

If we want to put this image into a screen rectangle with lower left corner $(l, b)^T$, width w and height h, all we need to do is to scale the coordinate up and add an offset. The matrix performing this so-called *window-to-viewport* map is

$$M_{\text{viewport}} = \begin{pmatrix} \frac{w}{2} & 0 & 0 & \frac{w}{2} + l \\ 0 & \frac{h}{2} & 0 & \frac{h}{2} + b \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Note that this matrix outputs homogeneous coordinates $(wx, wy, w)^T$; in a last step we need to de-homogenize the coordinates, resulting in 2D-coordinates $(x, y)^T$.

2.2 Colors

To explain lighting, we need first to understand what light is. Newton found out quite early that light is composed of many different color components; so to understand light, we will first have a closer look at what colors are, and derive the different color models that are important in the field of computer science.

After this we will be ready to look at local illumination, i.e. the calculation of light intensity or color at a given point of the scene. Note that lighting is calculated per-vertex; extending these values to lines and polygons is deferred to the shading stage of the rendering pipeline (see chapter 4.4).

2.2.1 Characterization of Colors

Color is an important property of materials, and it has been studied for quite a long time already. We will start this chapter by having a look at colors from different angles, adopted from different fields of science — arts, physics and biology. The way colors are represented in computer science is largely influenced especially by the perception of colors, so biology might be the most important field to look at.

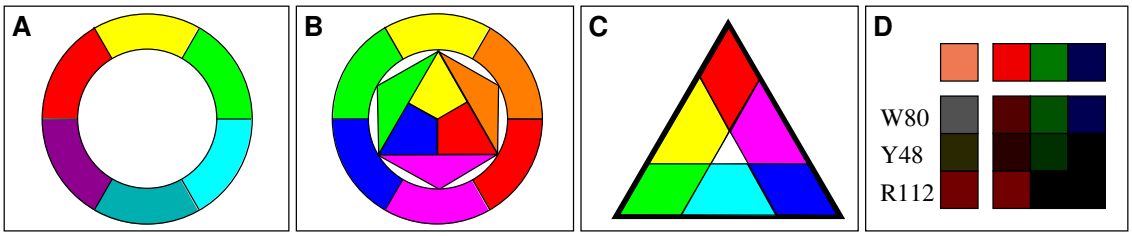


Figure 2.9: Colors in Arts: (a) Newton, (b) Itten, (c) Küppers, (d) Küppers decomposition.

Arts

Artists mostly look at color from the viewpoint of aesthetics, i.e. how colors stimulate humans. The first one who discovered colors was Newton in 1647; he used a prism to split the light into its spectral colors. In this more physical interpretation, he ordered the colors according to their wavelengths (cf. Fig. 2.9a).

The first person who studied the psychological effects of color was Goethe; he distinguished colors according to *temperature* (warm, cold colors), a classification still widely used nowadays. This classification was extended by Kandinski, who invented the notion of *color shapes* (round colors, sharp colors).

Itten classified colors according to the amount of *base components* they are mixed of. As basic colors he chose red, yellow, and blue (cf. Fig. 2.9b). This mixture system has been used for quite a long time, although it does not represent the way the human's eye senses colors.

The system used today is the one of Küppers. According to him, colors consist of the following components (cf. Fig. 2.9c):

- *Base Colors*: Red (R), Green (G), and Blue (B).
- *Secondary Colors* (mixtures of two base colors): Cyan (C), Magenta (M), and Yellow (Y).
- *Achromatic Colors*: Black (K) and White (W).

We illustrate this color model with the RGB color $(240, 128, 80)$, using color intensities from 0 to 255 (cf. Fig. 2.9d):

- Achromatic color: first, we subtract the minimum of all colors, giving the white component. In our example we obtain `W80`. The remaining color is (160,48,0). We also define the black portion to be $255 - \max\{R, G, B\}$ (in this example we get `K15`).
- Secondary color: from the nonzero components we again subtract the minimum. Thus, we subtract $(48, 48, 0)$, yielding $(112, 0, 0)$. Since we removed yellow (red+green), this component is `Y48`.
- Base color: the last nonzero component is the base color, here `R112`.

Thus, in Küpper's model, the RGB-color $(240, 128, 80)$ is `R112+Y48+W80`. We can calculate some aesthetic properties from this representation:

- *Achromaticity (Unbuntart)* is defined to be the quotient of the white and black portion:

$$W80/K15 = 5.33$$

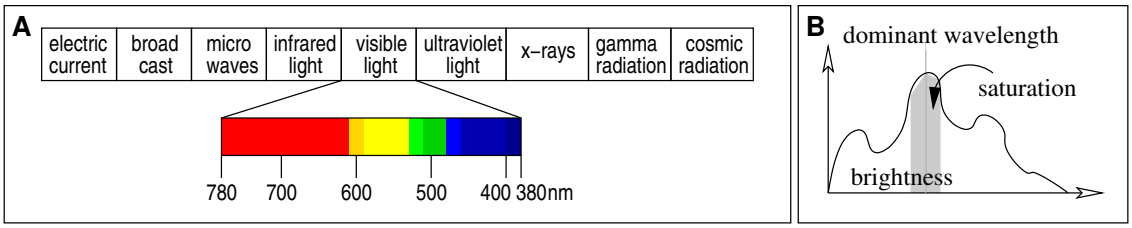


Figure 2.10: Colors in Physics: (a) electromagnetic spectrum, (b) spectral decomposition.

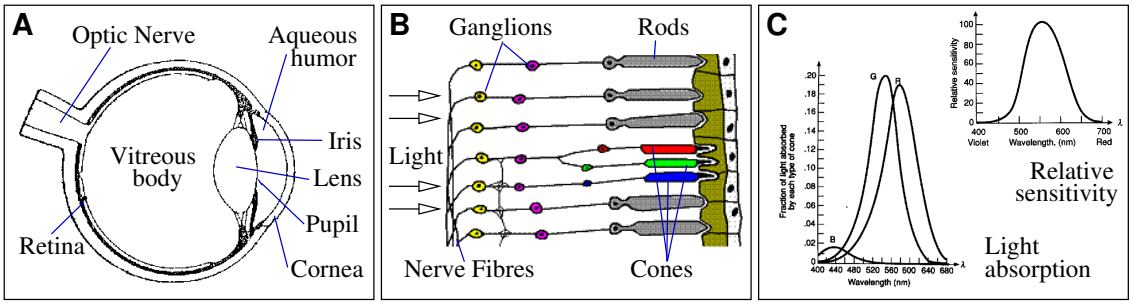


Figure 2.11: Colors in Biology: (a) eye, (b) retina, (c) response to light.

- *Chromaticity (Buntart)* is defined to be the quotient of the secondary and the primary portion:

$$Y48/R112 = 0.43$$

- *Color Degree (Buntgrad)* is defined to be the sum of the chromatic portions divided by the sum of the achromatic portions:

$$(Y48 + R112)/(W80 + K15) = 1.68$$

Physics

Colors can also be regarded as electromagnetic waves of different length. Visible wavelengths are those in between 380nm and 780nm, representing the whole color spectrum from violet, blue, green, yellow up to orange and red. Larger wavelengths constitute the invisible infrared light, microwaves, broadcast and electric current; ultraviolet light, x-rays, gamma- and cosmic radiation have smaller wavelengths (cf. Fig. 2.10a). A color can be regarded as a spectrum of wavelengths in the visible area. We can derive some notions from this spectrum (cf. Fig. 2.10b):

- *Hue* is the color corresponding to the *dominant wavelength*, i.e. the wavelength of maximum intensity in the spectrum.
- *Saturation* is the excitation purity of the dominant wavelength, i.e. the amount of the dominant wavelength compared to the white or grey component of the color. E.g., gray levels have saturation 0, while pure red has saturation 1. In case of full saturation the spectrum only has one wavelength and the color is said to be *spectral*.
- *Lightness*: the lightness (or brightness) is the amount of light energy corresponding to the area below the spectral curve.

Biology

Let us complete our excursion by looking at how a human's eye senses colors. Responsible for this sensation are receptors in the retina of the eye, of which there are two classes (cf. Fig. 2.11b):

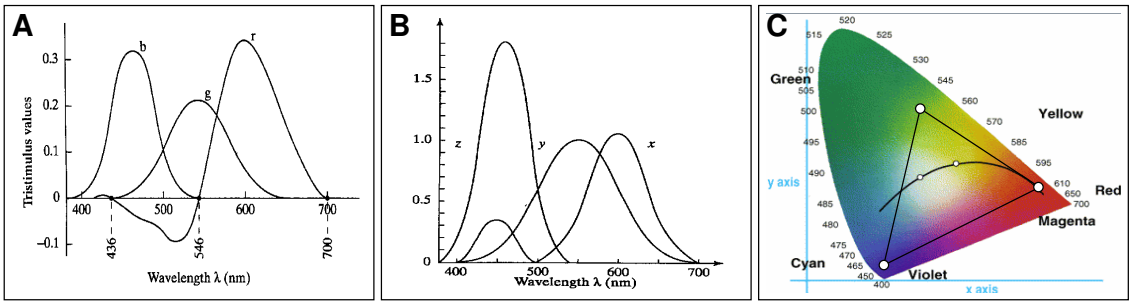


Figure 2.12: CIE Model: (a) RGB mix for visible colors, (b) CIE curves, (c) CIE chart.

- *Rods (Stäbchen)* are very sensible to light intensity, but they cannot distinguish colors.
- *Cones (Zäpfchen)* are receptors detecting color. There are three kinds of cones detecting red, green, and blue color, respectively.

The distribution of the receptors is not uniform: in the center of the retina there are few rods and many cones, enabling a good distinction of colors in the center of our view. On the contrary, at the boundaries of the retina there are only rods, therefore we can detect shapes in these regions, but no colors.

Fig. 2.11c depicts the light sensibility to different wavelengths of the red, green and blue cones (left) and of the eye in general (right). It follows that we best respond to medium wavelengths (green), and the response curve decreases towards lower and higher frequencies (blue and red). The reason for this fact is simply the distribution of cones; there are much more cones that can detect green than there are for red and blue.

The CIE Model

Since the human eye has separate receptors for red, green, and blue color, it seems to be natural to mix colors using these three base colors. Unfortunately, it is not possible to mix all colors by adding weighted contributions of red, green and blue. As Fig. 2.12a depicts, one would need negative weighting coefficients for the red component to represent wavelengths between $436nm$ and $546nm$. This is obviously impossible, since no device can emit negative amounts of light.

To alleviate this problem, in 1931 the *Commission Internationale d'Eclairage* tried to find three spectral distributions from which each visible color could be constructed using *positive* weighting coefficients. They found three completely artificial primary colors with this property, and these base colors were given the names $\mathbf{X}$, $\mathbf{Y}$, and $\mathbf{Z}$. Using these primary colors, any color C can be written as $C = X \cdot \mathbf{X} + Y \cdot \mathbf{Y} + Z \cdot \mathbf{Z}$ with $X, Y, Z \geq 0$ (cf. Fig. 2.12b).

These three colors build a base of the three dimensional CIE space. However, since adding the same relative amount to all color components just changes the brightness of the color, it is sufficient to normalize w.r.t. brightness. This can be done by restricting the CIE space to the plane $X+Y+Z = 1$, i.e. by mapping all colors (X, Y, Z) onto this plane by

$$x = \frac{X}{X+Y+Z},\ y = \frac{Y}{X+Y+Z},\ z = \frac{Z}{X+Y+Z} = 1 - x - y.$$

The resulting chromaticity values (x, y, z) are normalized in the sense that they sum up to 1, therefore it is sufficient to only specify the two values (x, y). The resulting CIE diagram is shown in Fig. 2.12c. In this figure you will find the three colors red, green and blue together with the triangle they span; only colors within this triangle can be displayed using the RGB model.

2.2.2 Colors in Computer Graphics

Now that we have discussed how we sense color, we can discuss their general use in computer graphics.

In order to generate reproducable images, we need some mechanism to *calibrate* monitors. We will also explain the need for *Gamma Correction*. Finally we will discuss several color models that are used to specify colors for monitors, printers or TV sets.

Monitor Calibration

Monitor phosphors do not always emit the same color: the color depends on the phosphor used and the voltage applied. Therefore, a monitor needs to be calibrated in order to display reproducable CIE colors.

Between the (R, G, B) values (scaled to $[0, 1]^3$) used to control the monitor's color intensity and the CIE values (X, Y, Z) there is a linear correspondence:

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} X_r & X_g & X_b \\ Y_r & Y_g & Y_b \\ Z_r & Z_g & Z_b \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

If we manage to find the values in this matrix, we know which (R, G, B) values we have to use to produce the CIE color (X, Y, Z). From the monitor's manufacturer (or from physical measurement) we know the (relative) chromaticity values (x_r, y_r), (x_g, y_g) and (x_b, y_b) of the red, green and blue monitor phosphor, respectively (we can find the z coordinates according to $z_i := 1 - x_i - y_i$).

Therefore we only have to find the intensities $C_i := X_i + Y_i + Z_i$, such that $X_i = x_i \cdot C_i$ $(i \in \{r, g, b\})$. In order to do this we need a reference color, say white. Given the CIE-position of the monitor's white color (X_w, Y_w, Z_w) we have

$$\begin{aligned} \begin{pmatrix} X_w \\ Y_w \\ Z_w \end{pmatrix} &= \begin{pmatrix} x_r C_r & x_g C_g & x_b C_b \\ y_r C_r & y_g C_g & y_b C_b \\ z_r C_r & z_g C_g & z_b C_b \end{pmatrix} \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \\ \iff \begin{pmatrix} X_w \\ Y_w \\ Z_w \end{pmatrix} &= \begin{pmatrix} x_r & x_g & x_b \\ y_r & y_g & y_b \\ z_r & z_g & z_b \end{pmatrix} \begin{pmatrix} C_r \\ C_g \\ C_b \end{pmatrix}. \end{aligned}$$

Solving this linear system yields the required C_i-values based on which we can calculate the color calibration matrix.

Gamma Correction

A difficulty in displaying colors or color differences on physical devices is that the human eye responds to relative instead of absolute color differences.

In order to verify this, one can pick three light bulbs with $50W$, $100W$ and $150W$. Although the brightness difference between them is the same ($\Delta = 50W$), the difference between the $50W$ and $100W$ bulb will appear larger than the difference between the $100W$ and the $150W$ bulb. This is simply due to the fact that the relative differences are not the same: in the first case the power doubles, in the second case it increases by only 50%.

The solution to this problem is a logarithmic scale for color intensities. We recompute the intensities of the colors such that the new values match our perception.

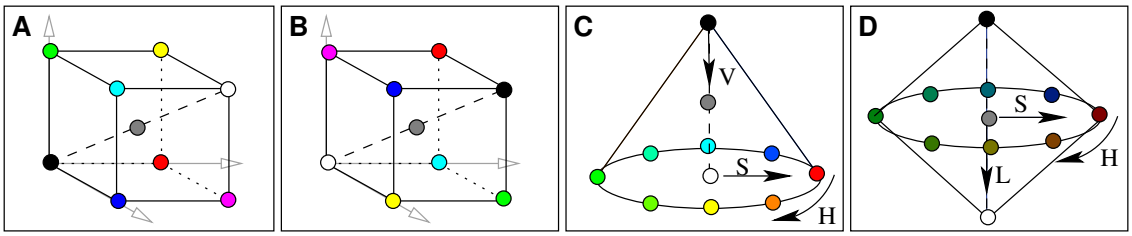


Figure 2.13: Color Models: (a) RGB, (b) CMY, (c) HSV, (d) HLS.

The first step towards this scale is to set a minimum intensity I_0 and a maximum intensity I_n. Thus, I_0 represents the black color and I_n represents white. The ratio $I_0 : I_n$ is the contrast ratio of the device. Whatever the real values for I_0 and I_n are, we will w.l.o.g. assume that $I_0 = \varepsilon > 0$ and $I_n = 1$.

What we want to achieve is that the ratio between two steps stays the same:

$$I_{j+1}/I_j = r = const.$$

Solving this recursion yields

$$I_j = r^j \cdot I_0, \quad r = \sqrt[n]{1/I_0}$$

Displaying these intensities on a monitor is not easy because of the non-linear correspondence between intensity I and applied voltage V:

$$I = k \cdot V^\gamma$$

with constants k and γ. In order to display a given intensity I we first find the closest I_j by

$$j = \text{round}(\log_r(I/I_0)), \quad I_j = r^j \cdot I_0.$$

The voltage V_j to be applied is then

$$V_j = \text{round}((I_j/k)^{1/\gamma}).$$

These voltages are usually stored in a look-up table. The use of this look-up table is referred to as *gamma-correction*, and is in general done by the graphics hardware or the monitor. It can be adjusted by the user by tweaking the value of γ, usually in the range $[2.2, 2.5]$.

Color Models

We have seen that it is very natural to stimulate the three kinds of receptors in a human's eye directly. Although it cannot represent all visible colors, the RGB model is most widely used. Virtually every display device creates images by addressing red, green, and blue phosphors behind the display mask. For printing, the CMY model is used; this is basically the inverse of the RGB model: instead of additively emitting light, light waves are subtractively being blocked. In some areas different representations of colors are used; here we will discuss the YIQ and the HSV system. Both systems allow for an efficient computation from and to the RGB model, and can thus be regarded as special cases of it.

RGB Model In the RGB-model colors are composed of the three components red, green, and blue. This results in a color cube as depicted in Fig. 2.13a, the principal axes being the three base colors. The origin is black, and the line for which all three values are the same is the main diagonal of the cube, representing gray values.

The RGB model is an *additive model*: based on black, colors are added until the target color is composed. This principle matches the way displays normally work; the background is black, and light is emitted according to the color intensities by addressing the phosphors with the corresponding voltage. For this reason, most displays are using RGB when displaying images, whatever the input format of the color might be.

CMY and CMYK Models The CMY model has three base colors cyan, magenta and yellow (cf. Fig. 2.13b), being the complementary colors of red, green and blue. In contrast to the RGB model, this model is *subtractive*, i.e. starting with white, pigments are added to block light wavelengths until the target color is created. Obviously this model comes quite handy for printers which have to print color on white paper.

There is a small technical problem with the CMY model: although mixing all colors with full intensity should yield black, the result is more a dark brown. The CMYK model therefore adds black as a fourth component. Based on the CMY color triplet (c, m, y), we just define k to be the minimum of the three components, and subtract it from the others. Thus, the new color quadruple (c', m', y', k') can be computed as follows:

$$k' = \min\{c, m, y\}, \quad c' = c - k', \quad m' = m - k', \quad y' = y - k'$$

YIQ Model The YIQ model aims to have one component representing the brightness of the color. Used widely in television broadcasting, this makes it possible to keep compatibility with black/white TV devices (they just ignore the two other components). Also, since the human's eye is not as sensitive to color information (hue) as to color intensity, this split makes it possible to use less resolution for chromaticity information.

The Y component encodes the brightness of the color, taking into account that the eye has different levels of perception for the three base colors. The two other components I and Q specify the chromaticity of the color. In detail, the conversion matrix is:

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.3 & 0.59 & 0.11 \\ 0.6 & 0.28 & 0.32 \\ 0.21 & 0.52 & 0.31 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

The HSV and HLS Models These models are more user-oriented; instead of being composed of three color components, in the HSV model a color is specified by its *hue* H, by its *saturation* S and its brightness or *value* V. Obviously, this is a much more intuitive way of defining colors, and it is used widely in graphics applications.

For the simple fact that saturation and hue get more important the brighter the color is (in the extreme case, black has neither hue nor saturation), the HSV-colors are normally depicted in a half cone (cf. Fig. 2.13c): H is the angle, S is the radius and V the height.

Actually, saturation and hue are also not very important for very bright colors. Therefore one can also represent the colors by a double cone, with black being one apex, and white being the other. The corresponding color model is the HLS model, representing *hue*, *lightness*, and *saturation* (cf. Fig. 2.13d).

2.3 Local Lighting

A very important part of the rendering pipeline is the computation of lighting, since it greatly enhances the realism of the rendered images. We already stated that lighting is calculated per

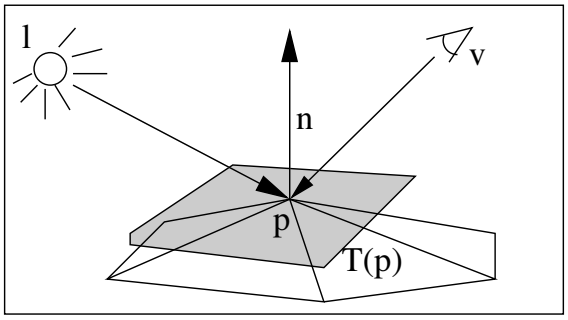


Figure 2.14: Lighting computation depends on position p, normal n, light source position l and viewing position v.

point, and that the transfer of these values into the interior of lines and polygons is performed in the shading stage of the rendering pipeline (see chapter 4.4).

The light intensity at a surface point depends on its position p, its normal n, the position of the light source l and the position of the viewer v (cf. Fig. 2.14). For that reason, we need the normals of surface points to do the lighting computations. But what happens to a point's normal vector if the corresponding point gets affinely transformed, e.g. by the modelview transformations?

A normal vector and a point define a tangent plane. Suppose we are given such a plane by its normal $n = (n_x, n_y, n_z)^T$ and its distance to the origin d. The plane can be written in implicit form (using the homogeneous representations $\mathbf{p} := (x, y, z, 1)^T$ and $\mathbf{n} := (n_x, n_y, n_z, d)^T$):

$$(n_x, n_y, n_z, d) \cdot (x, y, z, 1)^T = \mathbf{n}^T \mathbf{p} = 0$$

If we apply an affine transformation (represented by the matrix M) to points on this plane, the transformed points will still lie on a plane, but the respective normal vector and distance will change to $\mathbf{n}' := (n'_x, n'_y, n'_z, d')^T$. Therefore, we get the new plane equation $\mathbf{n}'^T (M\mathbf{p}) = 0$. To find the new normal and distance we reorder the terms:

$$\begin{aligned} 0 &= \mathbf{n}'^T (M\mathbf{p}) = \left(\mathbf{n}'^T M\right) \mathbf{p} = \underbrace{\left(M^T \mathbf{n}'\right)}_{=\mathbf{n}}{}^T \mathbf{p} \\ \Rightarrow \quad \mathbf{n}' &= \left(M^T\right)^{-1} \mathbf{n} =: M^{-T} \mathbf{n} \end{aligned}$$

So whenever we affinely transform a vertex by a matrix M, we have to transform its normal with the inverse transposed M^{-T}.

2.3.1 The Phong Lighting Model

Calculation of light intensity at a given point p with normal n is actually very complex; the reflected light is not just the direct illumination caused by light sources, but also light which has been reflected at other surface points into the direction of the point in question. Thus, for realistic effects, we need global information, which we do not have.

The Phong model is an approximation to the correct solution based on local information only. Specifically, in that model lighting is composed of three components. The total brightness of a surface point is the sum of these three components.

- *Ambient Color*: This component models the light inherent in a scene, i.e. it approximates the global light exchange of objects with each other. It models the fact that objects are almost never absolutely dark, even if no light source directly shines on them (cf. Fig. 2.15a).

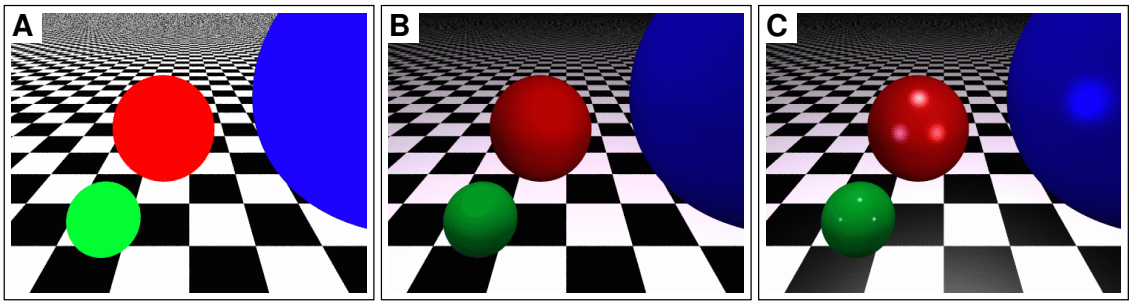


Figure 2.15: Phong Model: (a) ambient, (b) diffuse, (c) diffuse and specular.

- *Diffuse Color* models the reflection of light on matt or dull surfaces. Light arriving at the surface is equally reflected into all directions (cf. Fig. 2.16a). The light intensity depends on the angle in which the light hits the surface, but it does not depend on the viewer's position (cf. Fig. 2.15b).
- *Specular Color* models the (almost perfect) reflection of light we know from shiny surfaces. Most light rays are reflected in the same angle they hit the surface (cf. Fig. 2.16c). Therefore, the amount of specular reflection (i.e. how much light is reflected into the viewer's direction) depends not only on the position of the light, but also on the position of the viewer (cf. Fig. 2.15c).

The colors we will compute in the next subsections are all specified in the RGB model. However, all results are also valid for other color models.

Ambient Lighting

Ambient lighting models the light that is inherent in a scene, because of global effects such as indirect lighting. Since we cannot model these global effects using local illumination only, these effects are approximated by the ambient component. The ambient light is independent from the position of the viewer and the light sources; however, it does depend on material properties. Consider a pure green object: it can only reflect green light, i.e. pure red light, for example, would be absorbed, and if red were the only light source, we could not see the object.

We model the ambient reflectance of an object by a material matrix $\alpha_a \in \mathbb{R}^{3\times 3}$. This matrix is a diagonal matrix of the red, green, and blue ambient reflectivities, i.e. $\alpha_a = \text{diag}(\alpha_{a,r}, \alpha_{a,g}, \alpha_{a,b})$; these reflectivities are scaling factors out of $[0, 1]$. Multiplying this matrix by the global ambient intensity C_A yields the ambient lighting component (cf. Fig. 2.15a):

$$C_a = \alpha_a \cdot C_A$$

Diffuse Lighting

Diffuse lighting is a direction dependent term in the Phong model. It models the reflection occurring on rough or matt surfaces (cf. Fig. 2.15b): the incoming light is scattered uniformly into all directions (cf. Fig. 2.16a).

Therefore, the diffuse intensity at a given point is the amount of light energy hitting that point. Let us assume that a light ray has a diameter of one unit and carries an energy L_e per square unit. As Fig. 2.16b illustrates, the area A that is lit by the light ray evaluates to

$$A = \frac{1}{\cos\varphi},$$

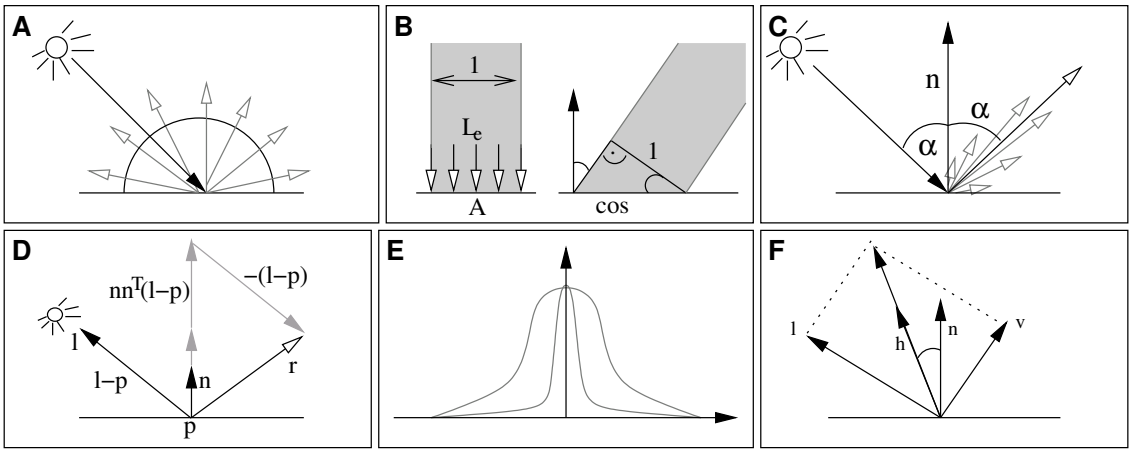


Figure 2.16: Lighting: (a) diffuse reflection, (b) light energy, (c) specular reflection, (d) reflected vectors, (e) shininess, (f) Phong-Blinn approximation.

where φ is the angle between surface normal and light ray. This means, the energy exposed to this square unit is

$$E = \frac{L_e}{A} = L_e \cdot \cos\varphi$$

The amount of diffusely reflected light is proportional to the cosine of the angle between surface normal and light direction (*Lambertian reflection*). For a light source at position l emitting the light C_l onto a surface point p with normal n and diffuse material property matrix α_d, we get for the diffuse component:

$$C_d(p, n, l) = \alpha_d \cdot C_l \cdot \cos\varphi = \alpha_d \cdot C_l \cdot \frac{n^T(l-p)}{\|l-p\|}.$$

Specular Lighting

The specular component models the reflection that occurs on shiny surfaces: if a light ray hits the surface with an angle φ to the surface normal, it is reflected with the same angle (cf. Fig. 2.16c). This reflected light vector can be computed as $r = (2nn^T - I)(l-p)$ (cf. Fig. 2.16d).

Intuitively, the closer the viewing ray $v - p$ is to this reflected vector r, the higher the specular intensity. As we deviate from r, the intensity gets smaller. This effect is again modeled by a cosine factor, this time taking the angle between r and $v - p$ into account.

$$C_{sp}(p, n, l, v) = \alpha_{sp} \cdot C_l \cdot \left(\frac{r^T(v-p)}{\|r\| \, \|v-p\|} \right)^s$$

Depending on the material (roughness/shininess) not all energy is reflected into the direction r, but some of the energy is still scattered. This is modeled by the *shininess* exponent s (cf. Fig. 2.16e): The higher s is, the less light is reflected into directions deviating from r. For large values of s, the light is almost completely reflected into the direction r (perfect/complete reflection). The shininess s therefore controls the size of the specular highlights (cf. Fig. 2.15c).

Phong-Blinn Approximation

Calculating the specular component using the above formula is computationally quite costly. Therefore, an approximation to this formula by Phong and Blinn is normally used. This ap-

proximation is based on the *halfway vector* h of l and v (cf. Fig. 2.16f):

$$h = \frac{(v+l)}{\|(v+l)\|},$$

where l is the normalized vector from the point p to the light source and v the normalized vector from p to the viewer.

The angle between h and n is a good approximation to the angle between r and v. Hence, we can simplify the formula for specular lighting as follows:

$$C_{sp}(p,n,l,v) \approx \alpha_{sp} \cdot C_l \cdot \left(n^T h\right)^s.$$

Using this approximation yields very similar visual results in most cases, but requires substantially less calculations.

2.3.2 Other Lighting Effects

Attenuation We have not yet discussed one important physical property of light: *attenuation*. It can be observed that light intensity decreases with increasing distance between surface and light source. We need to take this effect into account in order to create realistic images.

A light source emits light equally in all directions. At distance r from the light source the light's energy is distributed on a sphere of radius r. Since the surface of this sphere grows quadratically with increasing radius, the light intensity decreases quadratically as we move away from the light source.

Unfortunately, using quadratic attenuation together with the approximations inherent in the Phong model simply does not look well. Instead we will use an additional linear attenuation factor and get:

$$\mathrm{att}(p,l) = \frac{1}{att_{lin} \cdot \|p-l\| + att_{quad} \cdot \|p-l\|^2}$$

The two coefficients att_{lin} and att_{quad} give the relative weighting of the linear and quadratic attenuation terms, respectively.

Spotlight Another effect to be taken into account is that light can be directional. The light sources we considered so far emit light equally into all directions.

In case of a spotlight, a light source is associated with a direction d into which it emits the most light. Its intensity decreases with increasing deviation from that direction. Therefore, we need another factor, called *spotlight-factor*, that models this effect. Similar to the diffuse and specular term, the directional attenuation can be expressed in terms of a cosine, leading to

$$spot(p,l) = \left(\frac{d^T(p-l)}{\|p-l\|}\right)^f$$

Here, d denotes the direction of the spotlight, and f is a light source property similar to the specular shininess: the larger f is, the more spotty is the light source, i.e. more light is emitted into the direction d and less is emitted into other directions.

Depth Cueing If we want to model large outdoor scenes, we also have to take atmospheric effects into account. With growing distance to the viewer, the environment seems to merge with the atmosphere, i.e. the colors fade to some grey-blue. This effect is modeled by the so-called *depth cueing*: a gray-bluish atmosphere color C_{dc} is blended into the color from the previous steps, depending on the distance to the viewer (represented by the factor b):

$$C_{\mathrm{final}} = b \cdot C_{phong} + (1-b) \cdot C_{dc}.$$

2.3.3 Putting it all together

We can now put all the components we derived in the last chapters together and arrive at a formula that computes the total light intensity for a given surface point p with normal n, and a set L of light sources:

$$C_{phong}(p,n,v) = C_a + \sum_{l \in L} \left[spot(p,l) \cdot att(p,l) \cdot (C_d(p,n,l) + C_{sp}(p,n,l,v)) \right].$$

As mentioned in the last subsection, this color may finally have to be blended with a depth-cueing color.

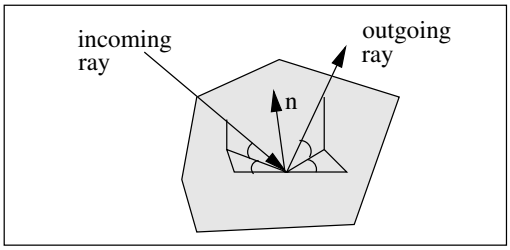
Using this formula we can achieve a very good approximation to realistic lighting, but have the advantage that the computation can be implemented quite efficiently, since only local effects are considered.

Further improvements in terms of efficiency can be achieved by

- *Directional light sources*: The light position is $(x, y, z, 0)$, corresponding to a light source infinitely far away or a light direction, respectively.
- *Infinite viewer*: The viewing position is assumed to be infinitely far away, i.e. v equals $(0, 0, \infty)$ and the viewing direction $p - v$ simplifies to $(0, 0, -1)$.

2.3.4 More Realistic Lighting: BRDF

Although the Phong model results in quite realistic images, there are still many effects not taken into account. To pick an example, surfaces with complex structures (e.g. a compact disc or metallic varnish) create anisotropic dazzling effects, which the Phong model will not be able to capture. The main reason is that the Phong model is isotropic, i.e. in this model the lighting only depends on the angle with the surface normal, but not on the angle in the tangential plane.



The so-called *BRDF* (*Bidirectional Reflectance Distribution Function*) models the correct behavior: it maps an incoming ray (defined by normal angle and tangential angle) and an outgoing ray (also defined by two angles) to the corresponding color $C(\varphi_{in}, \theta_{in}, \varphi_{out}, \theta_{out})$.

In the lecture „Computer Graphics II“ we will learn algorithms that approximate the BRDF much better than the Phong model does (cf. [Sar Dessai, 2003]). Of course, a higher degree of realism comes with higher computational costs, and the Phong model is the only model practically used that only requires local information and is fast enough to be implemented in the rendering pipeline.

Chapter 3

Line Rendering Pipeline

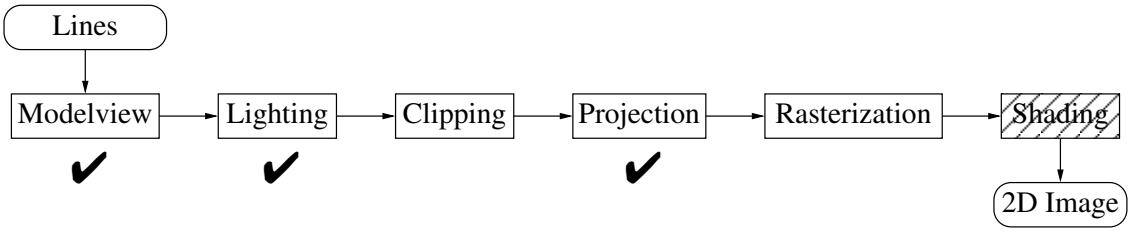


Figure 3.1: The Line Rendering Pipeline.

In the last chapter we dealt with the rendering of points. The sub-tasks to be performed were transformations (modelview and projection) and lighting computation. In this chapter we will step from zero-dimensional points to one-dimensional lines.

We will see in the next section that applying transformations to lines can be done by transforming the line's endpoints only. Therefore all we need to know about the transformation of lines has already been told in the last chapter.

Lighting computation is also performed for vertices only. In order to define color values for the interior points of lines, shading algorithms have to be applied. We will not discuss shading of lines here, but address this topic later when talking about shading polygons.

Clipping points was trivial since a point is either inside or outside the viewing volume. Lines, instead, can be completely inside, completely outside or partially inside the viewing volume. Hence, when processing lines we have to take a closer look at the clipping stage of the rendering pipeline.

The same holds for the rasterization stage: for points, just one pixel has to be set. Using lines, we have to determine a set of pixels building a discrete pixel approximation of a given continuous line.

3.1 Line Transformation

In this section we want to show that affinely or projectively transforming a line is equivalent to transforming its end-points and using the line defined by these transformed end-points.

Consider a line L defined by two points A and B, i.e. $L(\alpha) = (1-\alpha)A + \alpha B$. This line is to be transformed by an affine or projective 4×4 matrix M. We have to show that if we transform any

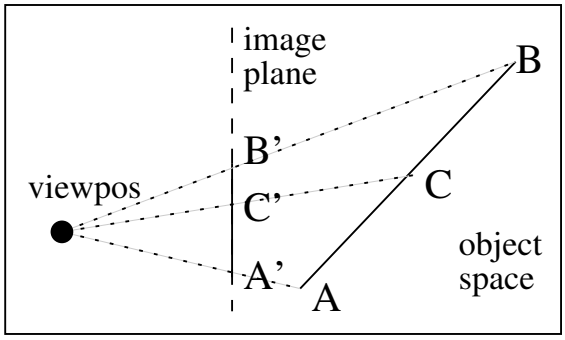


Figure 3.2: Projecting a line might change relative positions of C w.r.t. $\overline{AB}$ compared to C' w.r.t. $\overline{A'B'}$

point on the line, the transformed point lies on the line defined by the transformed end-points MA and MB, i.e.

$$\forall\alpha\ \exists\beta\ :\ ML(\alpha) = (1-\beta)MA + \beta MB. \tag{3.1}$$

If M is an affine transformation, this is trivially true since affine transformations are affinely invariant, i.e. the condition will be satisfied for $\beta = \alpha$.

In the case of projective transformations the situation is more complicated since in general β will not equal α, i.e. the relative positions on the line might change (cf. Fig. 3.2).

We can easily find a geometric verification that lines project to lines. The viewing position and the line span a plane in object space; the projection of the line is simply the intersection of this plane with the image plane.

But since we will need the conversion of α to β from Eq. 3.1 for the shading stage anyway, we will also derive this result mathematically.

A perspective transformation is composed of two steps: first we compute

$$(x', y', z', w')^T = M(x, y, z, 1)^T$$

and second we perform the de-homogenization

$$(x', y', z', w') \mapsto \left(\frac{x'}{w'}, \frac{y'}{w'}, \frac{z'}{w'}, 1\right).$$

Since matrix multiplication is a linear operator, Eq. 3.1 trivially holds for the first step. The critical part is the division by w', so we will concentrate on this operation.

Let the transformed end-points be $A' := MA$ and $B' := MB$ (the de-homogenization is still missing). If we restrict to the x coordinate, we have to show that we can find β such that

$$\frac{(1-\alpha)A'_x + \alpha B'_x}{(1-\alpha)A'_w + \alpha B'_w} = (1-\beta)\frac{A'_x}{A'_w} + \beta\frac{B'_x}{B'_w}$$

This can be derived as follows:

$$\begin{aligned}
\frac{(1-\alpha)A'_x + \alpha B'_x}{(1-\alpha)A'_w + \alpha B'_w} &= \frac{A'_x + \alpha(B'_x - A'_x)}{A'_w + \alpha(B'_w - A'_w)} \\
&= \frac{A'_x}{A'_w} + \frac{\alpha B'_w}{A'_w + \alpha(B'_w - A'_w)}\left(\frac{B'_x}{B'_w} - \frac{A'_x}{A'_w}\right) \\
&=: \frac{A'_x}{A'_w} + \beta\left(\frac{B'_x}{B'_w} - \frac{A'_x}{A'_w}\right) \\
&= (1-\beta)\frac{A'_x}{A'_w} + \beta\frac{B'_x}{B'_w}
\end{aligned}$$

Since this β does only depend on the homogeneous components, it is equally valid for the x-, y- and z-coordinate. Therefore, the de-homogenization maps a point $L(\alpha) = (1-\alpha)A + \alpha B$ on the line L to the point $L'(\beta) = (1-\beta)A' + \beta B'$ with the correspondence

$$\beta = \frac{\alpha B'_w}{A'_w + \alpha(B'_w - A'_w)}.$$

If we now consider the standard projection (or frustrum map) of a point p, we know that the homogeneous component of the point after the matrix multiplication — but before the de-homogenization — is set to $p'_w = -p_z$. As a consequence, we get the map from α to β by

$$\beta = \frac{\alpha B_z}{A_z + \alpha(B_z - A_z)}.$$

This proves that for the transformation of a line we can fall back on applying this transformation on the line's end-points only. This will finally result in a two-dimensional line specified by the two transformed and projected end-points.

3.2 Line Clipping

Up to now we discussed how to transform and project lines. Since lighting computations are performed per-vertex, the next topic we have to address is then *clipping* of lines. We want to discard lines (or line segments from now on) that are completely outside the viewing frustrum. Lines partially inside the frustrum have to be cut off accordingly.

In the rendering pipeline, clipping is performed after the frustrum transformation, i.e. after the frustrum has been warped into the unit cube $[-1,1]^3$. The advantage is that this clipping region is very simple, what greatly simplifies the design of the clipping algorithms. However, for reasons of simplicity, we will demonstrate the clipping algorithms for the 2D case only; extending them to 3D is straightforward.

The problem can therefore be stated as follows: given a 2D viewing rectangle on the screen and a set of lines, we need to find out which lines are completely inside the rectangle or completely outside. For those lines that are only partially inside the rectangle we need to find start- and endpoint of the segment inside the rectangle.

3.2.1 The Cohen-Sutherland Algorithm

The Cohen-Sutherland algorithm uses so-called *outcodes* to accelerate the clipping. These outcodes will help to identify the two trivial cases when the line is completely inside or outside the rectangle.

The clipping rectangle is bounded by the four border lines top (T), bottom (B), left (L) and right (R). Each of these lines separates two half-spaces, one that is "inside" and one that is "outside" (e.g. below/above the top line T). The clipping rectangle is the intersection of these four half-spaces.

The idea is to store this inside/outside information of a point p w.r.t. the four lines in a four bit *outcode*, such that the bits are 0 or 1 if the point p is inside or outside w.r.t. to the half-space bounded by T,B,R and L, respectively (in this order). This splits the screen area into nine regions, each of which is assigned a four bit outcode (cf. Fig. 3.3a).

Using these outcodes, we can easily detect whether a line is completely inside or completely outside the clipping rectangle. A line segment is completely inside if both end-points are inside. If both end-points are outside w.r.t. to the same boundary line, then the complete line is outside w.r.t. to this line. Let c_0 be the outcode for the starting point p_0, and c_1 the outcode for the end-point p_1. Then we detect the two trivial cases:

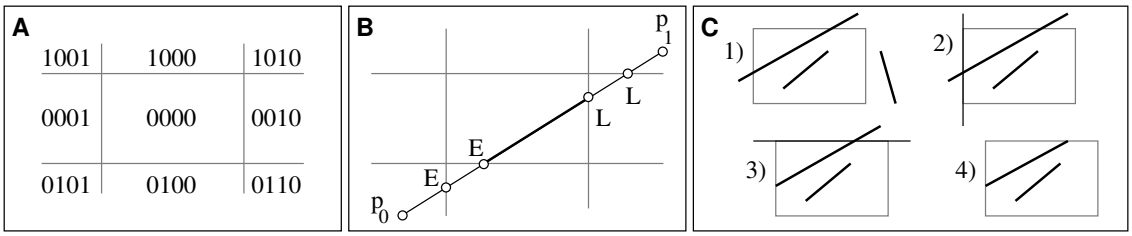


Figure 3.3: Line Clipping: (a) outcodes, (b) Liang-Barsky, (c) clipping procedure.

- *Trivial Reject*: if $(c_0 \,\&\, c_1) \neq 0$, the line is completely outside w.r.t. one border.
- *Trivial Accept*: if $(c_0 \,|\, c_1) = 0$, the line is completely inside.

If both of these tests fail, we really have to test for intersections of the line with the four boundary lines. For each of them, we first check whether the line crosses the boundary (check whether the corresponding outcode bit changes). If there is an intersection, we replace the outside end-point of the line with the intersection point and update the respective outcode. After we did this for all four boundary lines, the clipping is complete.

The advantages of this algorithm are clear: outcodes can be calculated easily, and based on simple logical operations on them we can detect lines which are completely outside or inside the clipping region. However, intersecting lines with the boundary is a problem; although we can use the outcodes again to determine the boundary lines to intersect with, we need to calculate up to four intersections and four new outcodes to get the final clipped end-points.

3.2.2 The Liang-Barsky Algorithm

The line clipping algorithm of Liang and Barsky uses a *parametric representation* of the lines, i.e. they represent a line between $p_0 = (x_0, y_0)$ and $p_1 = (x_1, y_1)$ as

$$L(\alpha) = (1 - \alpha)p_0 + \alpha p_1 \,,\ \alpha \in [0, 1].$$

For all boundary lines, the algorithm finds the parameter value α for the point at which the line crosses the boundary and updates the parameter interval accordingly, so that in the end we get the clipped line segment

$$L(\alpha) = (1 - \alpha)p_0 + \alpha p_1 \,,\ \alpha \in [\alpha_0, \alpha_1].$$

Since the boundary lines are either horizontal or vertical, computing the parameter value at the intersection is very easy. For instance, for the left boundary $x = l$ we obtain $\alpha = \frac{l - x_0}{x_1 - x_0}$.

The intersection parameter α is then used to update either the start parameter α_0 or the end parameter α_1, depending on whether the intersection point is *potentially entering* or *potentially leaving* the clipping rectangle (cf. Fig. 3.3b). The classification into entering or leaving intersection points is done by comparing the boundary's normal to the direction of the line $d = p_1 - p_0$ (the boundary normal is the perpendicular vector on the boundary pointing outwards). If they point into different directions (the dot product $n^T d$ is negative) the intersection is potentially entering, otherwise it is potentially leaving.

After clipping the line against all four boundaries and updating the parameter interval, we finally accept the resulting line segment if $\alpha_0 \leq \alpha_1$. In this case, $L(\alpha_0)$ and $L(\alpha_1)$ give the endpoints of the clipped line (cf. Fig. 3.3c). If $\alpha_0 > \alpha_1$ the line does not intersect the clipping rectangle and is discarded.

This algorithm can easily be extended to perform line clipping against any convex region; in that case the clipping region will be bounded by more than four lines and the intersection computations

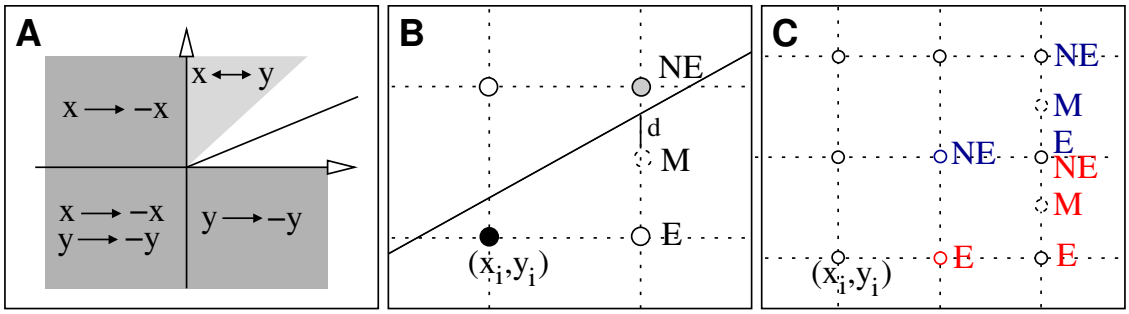


Figure 3.4: Line Rasterization: (a) octant restriction, (b)(c) Bresenham line drawing algorithm.

will be slightly more complicated. The adaptation to this more general case is known as the *Cyrus-Beck* algorithm.

3.3 Line Rasterization

Rasterization refers to the process of actually selecting and switching on all the pixels that belong to a primitive, in this case to a line. For a point, there is nothing to do here, since we have computed its pixel position already. In the case of lines we computed the pixel positions of the end-points only; we still need to know which pixels between these end-points belong to the primitive.

The main problem we have to solve is that we need to map continuous lines to discrete pixel positions, keeping the visual impression of continuity and the shape of a line. Since rasterization is performed for each primitive, it is essential to develop very efficient algorithms for that task.

In the following, the line's starting point will be $p_0 = (x_0, y_0)^T$, and its end-point will be $p_1 = (x_1, y_1)^T$. Let $\Delta x := x_1 - x_0$ and $\Delta y := y_1 - y_0$, then the slope of the line is $m := \Delta y / \Delta x$. We will limit the discussion to lines having a slope in between zero and one, and assume w.l.o.g. that $x_0 < x_1$. All other cases can easily be mirrored to this case by either changing signs or exchanging coordinates (cf. Fig. 3.4a).

Before we start with the rasterization, we recall the different representations of lines:

- *Parametric*: $y = mx + t$ with $m = \Delta y / \Delta x$ and $t = y_0 - mx_0$.
- *Implicit*: $ax + by + c = 0$ with $a = \Delta y$, $b = -\Delta x$ and $c = \Delta x t$.

3.3.1 First Ideas for Rasterization

A first try for line rasterization may be the following algorithm:

Algorithm 1 A first rasterization algorithm.

```
for (x=x0; x<=x1; ++x)
  setpixel(x, round(m*x+t));
```

This algorithm is very simple, but it has two major drawbacks: in each iteration it performs a multiplication and requires rounding $(mx + t)$ in order to get pixel coordinates (x, y).

Algorithm 2 Rasterization using DDA.

```
for (x=x0, y=y0; x<=x1; ++x, y+=m)
  setpixel(x, round(y));
```

We can solve the first problem by using *digital differential analysis* (*DDA*), i.e. by replacing multiplications by incremental additions. If we step from one iteration x to the next one $x + \delta x$, we compute the pixel coordinates

$$(x_{i+1}, y_{i+1}) = (x_i + \delta x, m(x_i + \delta x) + t) = (x_i + \delta x, y_i + m\delta x)$$

Therefore, instead of recomputing y_{i+1}, we increment y_i by $m\delta x$. Since δx will be one in all our cases, we get the following update rule:

$$(x_i, y_i) \mapsto (x_i + 1, y_i + m),$$

leading to the improved algorithm 2. However, although we managed to replace the multiplications by more efficient additions, we still require one rounding per iteration.

3.3.2 The Bresenham Midpoint Algorithm

Remember that we restricted to lines having slopes in $[0, 1]$. This implies that having placed a pixel at (x_i, y_i), the next pixel can be either North-East (x_i+1, y_i+1) (NE) or East (x_i+1, y_i) (E). We choose either E or NE depending on whether the line crosses above or beneath the midpoint $M := \frac{E+NE}{2}$ (cf. Fig. 3.4b). This can easily be checked by using the implicit representation of the line:

$$F(x, y) = ax + by + c$$

This function is negative for points above the line, and positive for those below the line. Therefore we can choose between E or NE by simply evaluating $F(M) = F\left(x_i + 1, y_i + \frac{1}{2}\right)$.

Since actually calculating $F(M)$ requires multiplications, we keep a decision variable $d = F(M)$, which is updated incrementally. We initialize d with the first midpoint:

$$F\left(x_0 + 1, y_0 + \frac{1}{2}\right) = a\left(x_0 + 1\right) + b\left(y_0 + \frac{1}{2}\right) + c = F\left(x_0, y_0\right) + a + \frac{b}{2} = a + \frac{b}{2},$$

since $F(x_0, y_0) = 0$, because the starting point is one the line.

In each step we decide for E or NE depending on the sign of d and update the decision variable d accordingly (cf. Fig. 3.4c). The two cases are:

- E is chosen. Then we update d as

$$d \mapsto F\left(x_i + 2, y_i + \frac{1}{2}\right) = a\left(x_i + 2\right) + b\left(y_i + \frac{1}{2}\right) + c = F\left(x_i + 1, y_i + \frac{1}{2}\right) + a = d + a$$

- NE is chosen. Then d gets

$$d \mapsto F\left(x_i + 2, y_i + \frac{3}{2}\right) = a\left(x_i + 2\right) + b\left(y_i + \frac{3}{2}\right) + c = F\left(x_i + 1, y_i + \frac{1}{2}\right) + a + b = d + a + b$$

This technique again replaces multiplications by incremental additions, and also avoids rounding to pixel coordinates. However, there is still one point we can improve: since d is initialized to $a + \frac{b}{2}$ we cannot use an integer variable for it. If we replace the implicit function $F(x, y)$ by $F' = 2F$ we still represent the same line, because the sign information does not change. But using F' now eliminates the fractions and reduces all calculations to integer arithmetic. Putting it all together results in the famous Bresenham algorithm 3.

Algorithm 3 Bresenham Algorithm.

```
a=y1-y0; b=x0-x1;
d=2a+b;
ΔE=2*a; ΔNE=2*(a+b);

for (x=x0,y=y0; x<=x1; ++x)
{
  setpixel(x,y);
  if (d<=0) { d+=ΔE; }
  else      { d+=ΔNE, ++y; }
}

setpixel(x,y);
```

Chapter 4

Polygon Rendering Pipeline

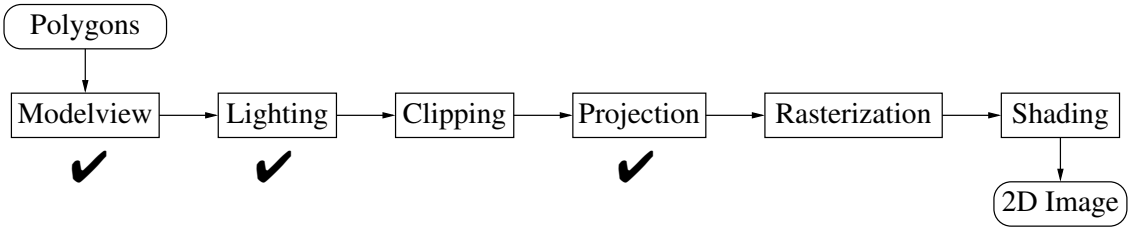


Figure 4.1: The Polygon Rendering Pipeline.

After having discussed the rendering pipeline for both points and lines, we now finally handle rendering polygons. Since almost every 3D model is composed of triangles or quads, polygons are the most important rendering primitives for the local rendering pipeline.

Using the analogous arguments as in the line case (see Sec. 3.1), we can show that for the transformation of polygons it is sufficient to transform its vertices only. Therefore the transformation and projection stages do not need to be discussed further.

For the clipping and rasterization stage we deal with algorithms that are similar to the line clipping and line rasterization methods we already discussed. However, since polygons are much more general than lines — they are two-dimensional primitives bounded by an arbitrary number of edges — these algorithms get slightly more complicated.

After cutting off all parts of the polygons outside the viewing volume (clipping) and determining the pixels corresponding to the projected 2D polygon (rasterization), we have to choose the right colors to fill these pixels with. Shading refers to transferring the vertex colors computed in the lighting stage to the interior of the polygons.

Finally, we will introduce texturing as a very effective and efficient method to enhance the visual detail of 3D models.

4.1 Polygon Clipping

Given an arbitrary polygon, we have to determine the part(s) of this polygon that are inside the viewing volume. As we may be facing non-convex polygons, too, the clipped results can consist of several components. Although polygon clipping can basically be reduced to clipping the polygon's edges, this task is slightly more complicated than clipping lines.

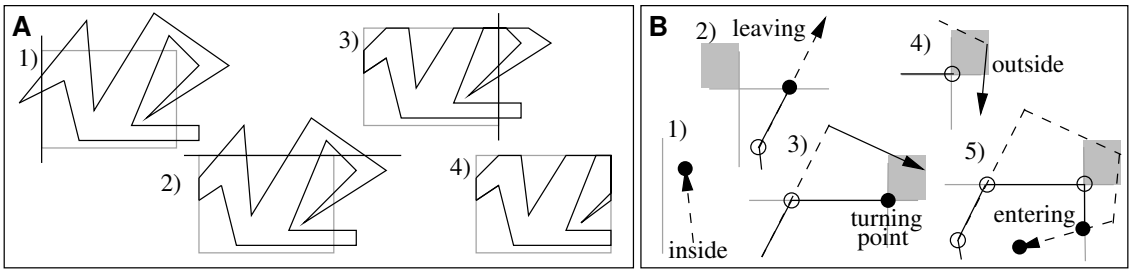


Figure 4.2: Polygon Clipping: (a) Sutherland-Hodgman, (b) Liang-Barsky.

4.1.1 Sutherland-Hodgman

The Sutherland-Hodgman algorithm is a generalization of the Cohen-Sutherland algorithm for polygon clipping. This method divides the problem of clipping a polygon against a clip rectangle into the simpler problem of clipping a polygon against an infinite line. Successively clipping the edges (p_i, p_{i+1}) of a given n-gon against the four lines T,B,R,L results in the desired clipping against the clipping rectangle.

In order to clip the polygon against one boundary, all edges of the polygon are traversed in a circular manner and clipped one after the other. Clipping an edge (p_i, p_{i+1}) outputs 0, 1, or 2 vertices, depending on whether and how this edge crosses the boundary:

- *Inside*: If the edge is entirely inside, we output p_{i+1}.
- *Outside*: If the edge lies completely outside, no output is generated.
- *Leaving*: If the edge leaves the clipping region (p_i inside, p_{i+1} outside) we output the intersection s of the boundary and the edge (p_i, p_{i+1}).
- *Entering*: If the edge enters the clipping region (p_i outside, p_{i+1} inside) we output the intersection point s as well as the end-point p_{i+1}.

Note that inside or outside refers to the current boundary line, not to the clipping rectangle. The classification into one of these four cases can easily be done based on the outcodes of the edge's end-points, analogously to the line clipping case. This edge clipping procedure is done for all polygon edges, and the clipped polygon is built from the output points that are connected in the order they have been generated. After clipping against one boundary, the resulting polygon is fed to the next clipper (B,R,L) in a pipelined fashion (cf. Fig. 4.2a).

The Sutherland-Hodgman algorithm is quite simple, since the above edge clipping procedure just has to be done for each polygon edge and clipping boundary. However, it has some drawbacks similar to the Cohen-Sutherland algorithm. We may compute intersection points that are discarded later, since they are outside another clipping boundary, leading to unnecessary calculations.

Also note that parts of the resulting clipped polygon might be degenerated: if the clipping procedure splits the polygon into several parts, these parts will still be connected by infinitely thin edges along the viewport boundary. This has to be taken care of in the rasterization stage (see Sec. 4.2).

4.1.2 Liang-Barsky

This algorithm is an extension to the Liang-Barsky algorithm for lines, because it applies the Liang-Barsky line-clipping method to each of the polygon's edges. Using the same case distinction as in the last section, we possibly add the end-point of the clipped edge to the vertex list of

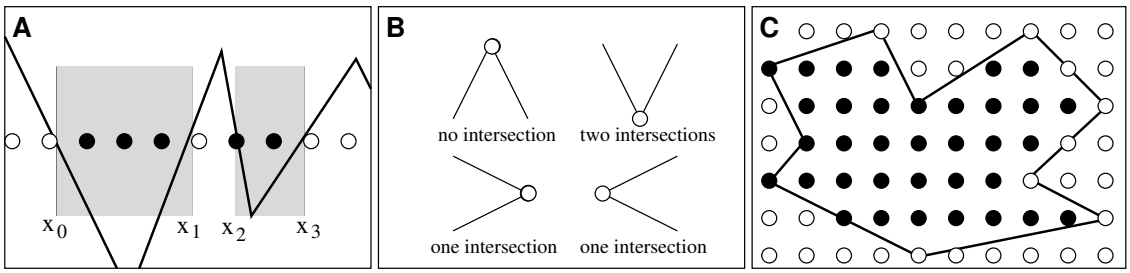


Figure 4.3: Polygon Rasterization: (a) scan-line, (b) pixel intersections, (c) result.

the clipped polygon. The difference to the Sutherland-Hodgman algorithm is that the two loops (clipping edges and polygon edges) are exchanged, i.e. here we clip each polygon edge against the viewing rectangle at once.

However, there is one special case that has to be taken care of. If, e.g., the polygon leaves the clipping rectangle through the left boundary and enters it again through the top boundary, we have to add the top-left boundary point to the list of output points, otherwise the top-left corner would be chopped off. More general, if the polygon passes through a corner region (the four regions that are outside w.r.t. *two* boundaries), we have to add the corresponding corner point to the output polygon (cf. Fig. 4.2b).

4.2 Polygon Rasterization

Rasterizing a polygon is obviously more complex than rasterizing a line. Apparently, a polygon is a 2D primitive, and therefore needs to be filled. This implies that we need to know which pixels lie inside the polygon. Checking this for each pixel in the convex hull is too expensive, so we need to find a faster way to do that.

Another problem occurs when two polygons share a common edge. We need to ensure that pixels along that edge are assigned to either the one or the other polygon, so that pixels are not set twice. As we will see, this can be solved by using consistent rules about which boundary pixels of a polygon belong to it, and which do not.

Note that optimized algorithms exist for triangles (as they are always convex). However, we will not present them here.

4.2.1 Scan-Line Conversion

The scan-line conversion algorithm traverses all scan-lines y (horizontal pixel lines) which intersect the polygon $(p_0, p_1, \ldots, p_n)$, i.e. for which holds

$$\min_i \left\{ (p_i)_y \right\} \leq y \leq \max_i \left\{ (p_i)_y \right\}.$$

For each of these scan-lines we have to compute the horizontal spans inside the polygon. This is done by intersecting the current scan-line with the polygon edges and sorting these intersection points by increasing x-component, yielding $x_0, x_1, \ldots, x_k$. Since the first intersection point is entering the polygon, the second one leaving, etc., we fill the spans between odd and even intersections (cf. Fig. 4.3a):

$$[x_0, x_1],\ [x_2, x_3],\ \ldots,\ [x_{k-1}, x_k]$$

This basic idea is very simple. However, we have to handle some special cases and use some optimizations for this algorithm to work correctly and efficiently.

Which pixels are inside?

When drawing polygons sharing a common edge we want to avoid overlaps as well as gaps between these polygons, i.e. each pixel should be set only once. This problem can be solved by defining the horizontal spans to be half-open on the right. After computing the (floating point) intersections $x_0, \ldots, x_k$ we round them to the *inside*-integers, and we also exclude integer intersection points on the right of an interval.

If the scan-line happens to intersect the polygon exactly at a vertex, we have to define how often we insert this intersection point into the list. Defining the polygon's edges to be half-open on the top solves this problem: a vertex intersection counts only if the vertex is the lower end-point of an edge. This results in zero, one, or two intersection outputs for a vertex intersection (cf. Fig. 4.3b).

The last special case we have to handle are horizontal polygon edges. They intersect a scan-line in infinitely many points. Using half-open edges already solves this point and we can just omit horizontal edges, i.e. they result in zero intersection points.

Fig. 4.3c shows the result of rasterization of a polygon using these rules.

Edge Coherence

Computing the intersections of the current scan-line with the polygon edges must be done cleverly, otherwise the algorithm will be impractically slow.

First of all, we need to determine the set of edges that actually cross the current scan-line y. Obviously, these edges are characterized by having y in their y-span, i.e. $y_{min} \leq y \leq y_{max}$. Edges crossing the current scan-line are called *active*. It is straightforward to keep two ordered edge lists to efficiently activate and deactivate edges:

- *Passive List*: the passive list contains all inactive edges and is ordered w.r.t. the y_{min} value of the edges; initially this list contains all edges. Whenever we change the scan-line from y to $y+1$, we move all edges with $y_{min} = y+1$ to the active list.
- *Active List*: the active list contains all edges which cross the current scan-line; initially this list is empty. It is ordered according to y_{max}. After changing the scan-line to $y+1$, we remove edges from the list that have $y_{max} = y$.

Since an edge intersecting the scan-line y will probably also intersect the next scan-line $y+1$, we can use incremental methods in order to accelerate the computation of edge intersections. If the x-intersection on scan-line y was x_y, the intersection at $y+1$ will be

$$x_{y+1} = x_y + 1/m \text{ with } m := \frac{y_{max} - y_{min}}{x_{max} - x_{min}}.$$

Therefore each element in the active edge list stores y_{max}, the current intersection x, and the increment $1/m$. Then this algorithm makes use of two kinds of coherence: *edge coherence* for the intersection computation and *scan-line coherence* when filling the x-spans.

Note that for triangles (instead of general n-gons) the scan-line conversion is much simpler, since triangles are always convex, leading to only two intersections points. Since there are just three edges (that can be pre-sorted) no sorting of intersection points is necessary. Hence, it is in general more efficient to first convert n-gons into $n-2$ triangles and to process the resulting triangles.

4.3 Triangulation of Polygons

As mentioned previously, triangles can be processed much more efficiently than general polygons. In fact, modern graphics APIs like OpenGL tessellate each n-gon into $n-2$ triangles before sending them through the rendering pipeline. Therefore we will discuss some triangulation algorithms here.

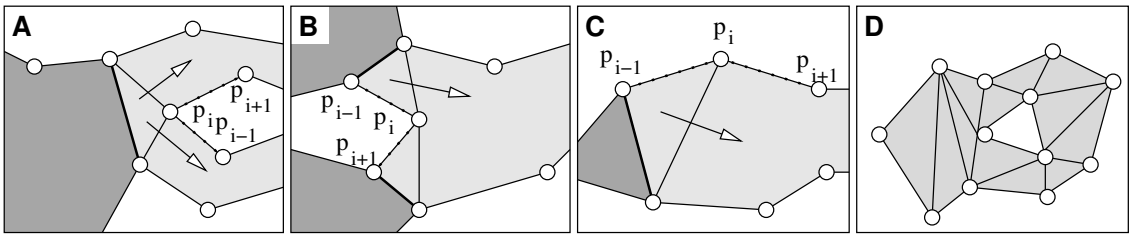
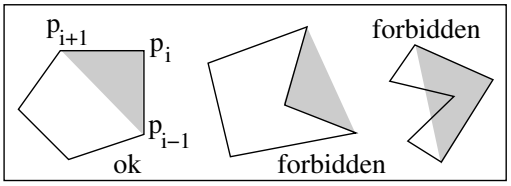


Figure 4.4: Sweepline Algorithm: (a) split, (b) merge, (c) generate, (d) result.

4.3.1 Marching (Corner Cutting)

A very simple triangulation method is to sequentially cut off corners until only one triangle is left. Cutting off a corner means selecting a vertex p_i and inserting an edge between its neighbors p_{i-1} and p_{i+1}; this results in the triangle $\Delta(p_{i-1}, p_i, p_{i+1})$ being removed from the polygon (see figure below).

When we cut off a corner we must take care that the triangle we cut off lies completely inside the polygon. This implies two conditions we need to ensure: the chosen vertex must be a convex one, and no other vertex may lie in the triangle (see right figure).



The convexity of the corner can be verified by using the 2D cross product $(p_{i-1} - p_i) \times (p_{i+1} - p_i)$ of the adjacent edges; the 2D cross product is defined to be

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \times \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} := x_1 y_2 - x_2 y_1.$$

The sign of this cross product specifies whether the corner is convex or concave; which sign corresponds to convexity is not yet clear, because we do not know whether the vertices are specified in clockwise or counter-clockwise order. To find out, we choose the vertex with minimum x-coordinate: this vertex must be convex. Calculating its cross product thus gives us the sign of convex corners.

To ensure that no point lies in the cut-off triangle, we need to check whether any of the points which do not constitute one of the triangle's vertices is inside it. In fact, it suffices to check this property for concave vertices only.

Now that we know how to check which triangles we may cut off, the algorithm is straightforward: we just loop through the polygon vertices in a circular manner, and whenever we find a valid vertex, we cut off the corresponding corner/triangle. We repeat this procedure until we cut $n - 3$ triangles of the n-gon (what remains is the last of the $n - 2$ triangles). The worst case behavior of this algorithm is $\mathcal{O}(n^2)$.

If we would like the triangles we cut off to be "well-shaped" we can first calculate the set of vertices that may be removed, calculate some quality measure for each of the triangles, and then cut off the best one.

4.3.2 Sweepline Algorithm

The sweep line algorithm traverses the vertices of the polygon from left to right, and joins three consecutive vertices to a triangle. In the ,,Computer Graphics II" lecture we will introduce a

sweepline algorithm that creates a Delaunay triangulation[1]; the algorithm we present here does not guarantee the Delaunay property, but it is a bit easier and suffices for now.

The first step is to order the vertices by their x-coordinate. Also, we keep a tuple of two vertices and initialize it with the first two elements in the list. The algorithm then proceeds as follows: we take the top element from the list, and create a triangle with this vertex and the two vertices in the tuple. Then, we replace the first element in the tuple with the second one, and the second one with the vertex we just added. This procedure is iterated until the last vertex has been processed.

We must take care of two special cases: the sweep front might split into two fronts, and two fronts might join. Fortunately, these special conditions can be checked by looking at the orientation of the adjacent edges. Let $p_i = (x_i, y_i)^T$ denote the vertex to be added, and let p_{i+1}, p_{i-1} denote the adjacent vertices in the polygon, respectively. Then we can distinguish three cases:

- Split: if $x_i < x_{i-1}, x_{i+1}$, we generate a new triangle, and then break the front up into two fronts (cf. Fig. 4.4a).
- Merge: if $x_i \geq x_{i-1}, x_{i+1}$, we are in a situation where two fronts need to be merged (cf. Fig. 4.4b).
- Add: if $x_{i-1} < x_i < x_{i+1}$, we simply need to add a new triangle (cf. Fig. 4.4c).

This algorithm has a complexity of $\mathcal{O}(n \log(n))$, if we assume that the number of fronts we get is limited by a constant factor. Although computationally better than the Marching algorithm, this algorithm is normally not used in practice; it is more complex, and since most polygons will have only few vertices, the computational advantage does not pay off.

4.4 Polygon Lighting and Shading

The last step in the rendering pipeline is shading. In this step, the pixels occupied by a primitive (as calculated during rasterization) are filled with color. During the lighting stage we computed color values for the vertices only. For point rendering this is all we need, but in case of lines and polygons we have to calculate colors for all interior pixels of the primitive, too.

In our discussion we will consider triangles only; lines can be shaded by applying the same rules (since they are degenerated triangles), and polygons can be reduced to triangles, as discussed in the previous chapter.

4.4.1 Lighting

For the lighting of points we assumed that we were given a normal vector for each point. In the case of triangle meshes, we can also derive vertex normals from face normals.

For a given face $\Delta(A, B, C)$ the normal vector is defined by the cross product

$$n_{\Delta(A,B,C)} := \frac{(B - A) \times (C - A)}{\|(B - A) \times (C - A)\|}.$$

The normal vector of a vertex can now be computed to be the weighted average of the normals of all adjacent faces Δ_i:

$$n_p := \frac{\sum_i \alpha_i n_{\Delta_i}}{\|\sum_i \alpha_i n_{\Delta_i}\|}.$$

[1] Among all possible triangulations, the Delaunay Triangulation maximizes the minimal inner angle of the triangles, leading to well shaped triangles.

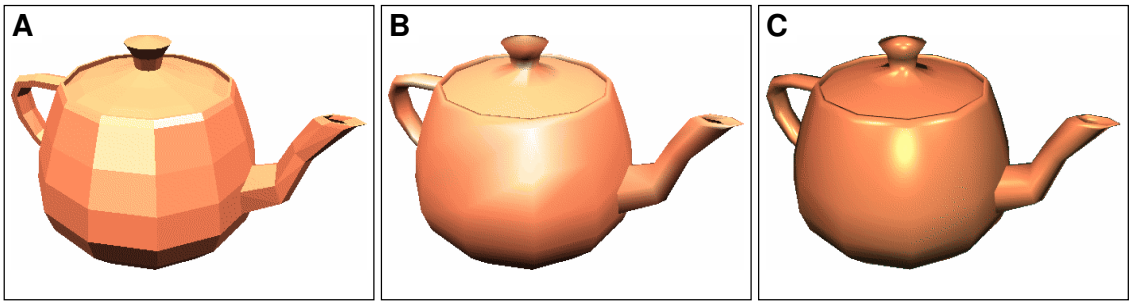


Figure 4.5: Shading models: (a) flat shading, (b) Gouraud shading, (c) Phong shading.

The weights α_i can be uniform, i.e. $\alpha_i = 1$, or they can correlate to triangle area or triangle opening angles at p. The result is a normal vector for each vertex that can be used to compute the Phong lighting model.

4.4.2 Shading Models

There are several shading models, which differ in complexity and visual realism. When we discuss how to shade a triangle, we assume $A = (x_a, y_a)$, $B = (x_b, y_b)$ and $C = (x_c, y_c)$ to be the pixels corresponding to the vertices of the triangle, and C_A, C_B and and C_C to be the colors of these vertices, as computed during the lighting stage. Let $p = (x, y)$ be the pixel in the triangle which we want to compute the color for.

Flat Shading

This is the simplest shading model: all pixels within the triangle are constantly colored with the average of the vertices' colors (cf. Fig. 4.5a):

$$C(p) = \frac{C_A + C_B + C_C}{3}.$$

One can also use the color value resulting from lighting the triangle's barycenter using the triangle's normal vector:

$$C(p) = \text{light}\left(\frac{A + B + C}{3},\ n_{\Delta(A,B,C)}\right).$$

It is obvious that with flat shading the triangulation of the object gets visible, especially for objects with high curvature. Unfortunately, the color differences are amplified by a perceptual effect called *Mach Banding*: the brain adds contrast at color boundaries, thereby increasing the perceived difference between the triangle faces.

Gouraud Shading

For the Gouraud shading model the pixel's color is interpolated based on the colors of the vertices. More precisely, the vertex colors are weighted according to the pixel's barycentric coordinates w.r.t. the triangle $\Delta(A, B, C)$:

$$C(p) = \alpha C_A + \beta C_B + \gamma C_C,$$

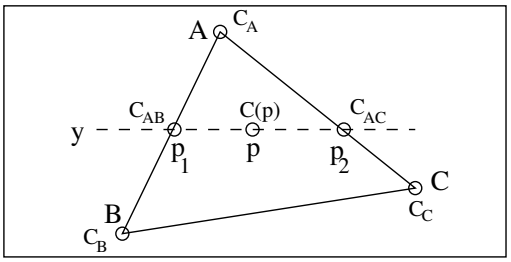
with $p = \alpha A + \beta B + \gamma C$ and $\alpha + \beta + \gamma = 1$.

Performing this bilinear interpolation based on barycentric coordinates can be decomposed into two linear interpolation steps: first we calculate the two intersections $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$ of the current scan-line with the triangle's edges and linearly interpolate the respective vertex colors

along the triangle edges. In a second step we linearly interpolate the two colors of the intersection points along the scan-line to get the final pixel colors.

These intersection computations and linear interpolations can easily be integrated into the rasterization stage. For the situation depicted in the figure we get

$$
\begin{aligned}
C_{AC} &= C_A \cdot \frac{y - y_c}{y_a - y_c} + C_C \cdot \frac{y_a - y}{y_a - y_c} \\
C_{AB} &= C_A \cdot \frac{y - y_b}{y_a - y_b} + C_B \cdot \frac{y_a - y}{y_a - y_b} \\
C(p) &= C_{AC} \cdot \frac{x_2 - x}{x_2 - x_1} + C_{AB} \cdot \frac{x - x_1}{x_2 - x_1}
\end{aligned}
$$



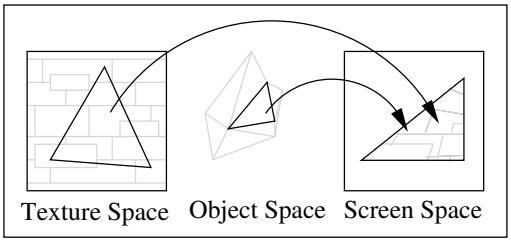
Although yielding much smoother results than flat shading (the triangle edges cannot be detected that easily), Gouraud shading still has an important flaw: if a highlight is in the center of a triangle and does not affect any of its vertices, the highlight will not get visible at all since only the dark colors of the vertices are interpolated (cf. Fig. 4.5b).

Phong Shading

Phong shading is the most complex shading model: instead of bi-linearly interpolating color values we interpolate the normal vectors of the vertices. Hence, during rasterization we are provided with a normal vector for each pixel and can do lighting computation on a per-pixel basis. This obviously leads to very expensive computations since now we compute the Phong lighting model for each pixel instead of for each vertex. The quality of this shading model is, however, far superior to Gouraud shading, especially for coarse triangulations (cf. Fig. 4.5c).

4.5 Texturing

Texturing is a very common technique used for increasing the visual detail of objects. The idea is to paste (parts of) an image onto triangles. In this way the resolution of visual details corresponds to the texture resolution, instead of the vertex resolution. Hence, we can increase the visual detail of objects while keeping their geometric complexity rather coarse.



We will first discuss how to map an image (a texture) onto *one* triangle only, and then describe methods for texturing whole triangle meshes.

Although normally textures are 2D images, there are cases in which 1D or even 3D textures are suitable and we will discuss these types of textures as well.

Textures need not necessarily contain image data. *Reflection maps* (also called *environment maps*), for example, contain pre-computed reflections, which — if correctly mapped onto the object — are an easy means to simulate reflective surfaces.

One problem with textures is that sometimes many texture pixels (called *texels*) are being mapped onto one screen pixel, or — conversely — one texel is mapped onto many screen pixels. *Antialiasing* algorithms take care that in these cases the rendered image still looks well.

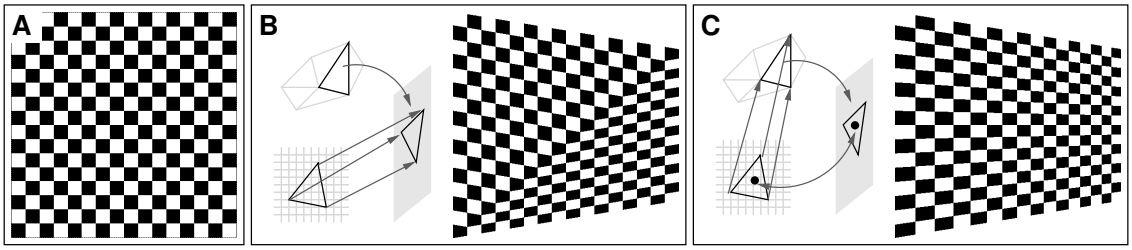


Figure 4.6: Texturing: (a) texture, (b) screen space interpolation, (c) object space interpolation.

4.5.1 Texturing Triangles

In order to texture a triangle we first have to select the region of the texture image to be mapped onto the triangle. To incorporate this, we just add two more scalars to each vertex, specifying the 2D position of the vertex in the texture. Thus, a vertex is now a quintuple

$$V = (v_x, v_y, v_z, t_u, t_v)^T.$$

These texture coordinates fully specify an affine map from the texture space triangle to the object space triangle. For the rendering this object space triangle is projected onto the image plane, yielding a screen space triangle.

Using the texture coordinates of the vertices, one can compute the texture coordinates of each pixel in the triangle by bilinearly interpolating these 2D coordinates. As depicted in Fig. 4.6, there are two possibilities for the interpolation to be done:

Screen Space Interpolation We can put the texture onto the triangle after having projected it onto the image plane. This would correspond to interpolating the texture coordinates over the screen space triangle.

However, this method has a huge flaw: as we have seen in Sec. 3.1, a projective map will in general change the barycentric coordinates of a point. Interpolation of texture coordinates in screen space does not take this perspective foreshortening into account, clearly visible at the checkerboard texture in Fig. 4.6b.

Object Space Interpolation To get correct results we have to interpolate the texture coordinates in object space. Each pixel in screen space is transformed back to object space, yielding a 3D point on the triangle. The pixel's texture coordinate is then interpolated based on the barycentric coordinates of the 3D point. Alternatively, one could integrate the perspective division into the rasterization replacing the linear interpolation. This basically corresponds to the map between α and β in Eq. 3.1 (Sec. 3.1).

This method is much more expensive since it requires one perspective division for each pixel. However, it results in perspectively correct texturing (cf. Fig. 4.6c).

4.5.2 Texturing Polygon Meshes

Up to now we only discussed the map of a texture onto one triangle. If we consider a whole object represented by a triangle mesh, we have to map all its vertices onto the texture (note that each vertex has only one texture coordinate, but may be shared by several triangles).

Since the object is three- and the texture is two-dimensional, this, in general, introduces distortion. Obviously, the aim is to compute a map such that this distortion gets minimal. There are two

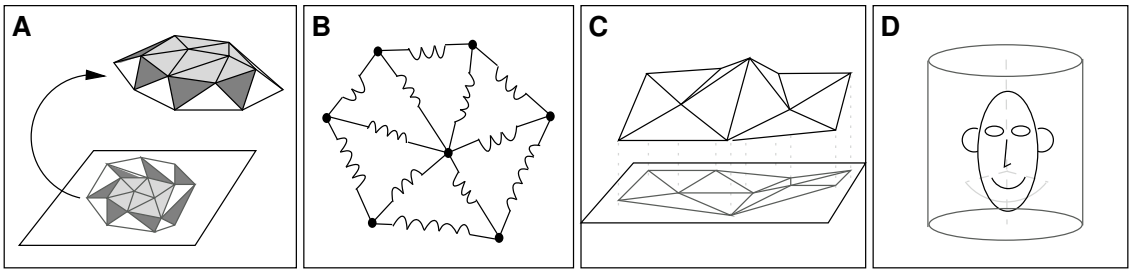


Figure 4.7: Parameterization: (a) problem, (b) spring mass model, (c) height map, (d) cylinder map.

main aspects to be considered for creating *low distortion maps*. On the one hand, one might want to preserve lengths and areas, and on the other hand one may aim at the preservation of angles. Keeping both properties is generally not possible, so one either selects one of these criteria or specifies a weighted average of both.

One way to compute a low-distortion map is using the so-called *spring mass model* (cf. Fig. 4.7b). Just to grasp the idea, one can replace the edges with springs with a force being inversely proportional to the length of the 3D edge. Then, one fixes the boundary vertices to some positions on the texture (it is possible to devise optimal positions for them). The positions of the spring vertices in balance are the optimal texture coordinates for the corresponding vertices. We will discuss low distortion maps in detail in ,,Computer Graphics II“.

There are simpler ways which might not be optimal, but still yield plausible results.

A very simple method is to consider the object to be a height field over some plane in object space, i.e. we interpret each vertex position as a two dimensional point (u, v) on the plane and a height $h(u, v)$ (cf. Fig. 4.7c). This plane should be a good approximation to the 3D geometry in order to avoid too much distortion. For example, one could use the best approximating plane in the least squares sense. This works well for approximatively planar objects, but the higher their curvature, the worse this method gets.

Another possibility is to approximate the object with some simple geometry. For example, one can wrap a cylinder around the object (cf. Fig. 4.7d), and then cast rays from the cylinder's axis through the object's vertices onto the cylinder's surface. Unrolling the cylinder then gives the 2D texture coordinates. Of course, high distortion occurs at places where the object is not approximately cylindric.

4.5.3 1D and 3D Textures

In the previous sections we just considered 2D textures. However, sometimes it is convenient to use 1D or 3D textures instead (cf. Fig. 4.8a).

When using 1D textures, the texture coordinate is a one-dimensional scalar $t_u \in [0, 1]$, in the case of 3D textures, the texture coordinate is a three-tuple $(t_u, t_v, t_w) \in [0, 1]^3$. 1D textures are often used to display iso-lines on objects; by assigning, e.g., temperatures to vertices and creating a 1D texture with the color scale, one can easily create a temperature map for the object.

3D textures can be used to create realistically structured surfaces like, e.g., wooden ones: just create a 3D texture with the year-rings of a tree and use it (cf. Fig. 4.8a).

In both cases the interpolation of texture coordinates works analogously to the 2D texture case, just with the difference that 1D or 3D texture coordinates are to be interpolated.

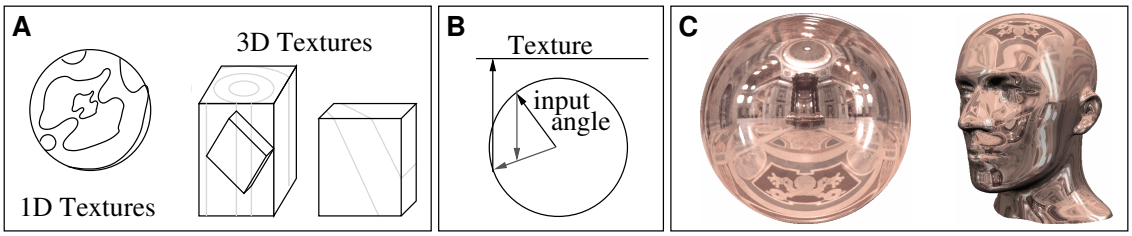


Figure 4.8: Special Textures: (a) 1D and 3D textures, (b)(c) environment mapping.

4.5.4 Environment Maps

Special texture maps can also be used to pre-compute the (approximate) reflection of the environment on the object's surface [Blinn and Newell, 1976, Miller and Hoffman, 1984]. Applying these so-called *environment maps* or *reflection maps* results in objects that realistically reflect their environment, like e.g. the liquid robot in Terminator 2.

Environment maps contain the 3D scene as seen from the center of the object. The first step is to approximate the object with a simple geometry like a cube or a sphere, and place it in the object's center. Then, viewing rays are emitted from the center into all directions, and the colors of the points in the scene hit by the rays are stored in the environment map. When shading the object, we can simulate reflections by looking up the corresponding entry in the reflection map.

Storing the color values corresponding to ray directions in the environment map requires a proper parameterization of the proxy object. We will discuss this for the case of *spherical* environment mapping. In this case the texture image looks like a photograph of a reflective sphere placed in the environment.

The corresponding parameterization is as follows: given a normalized direction vector d, we first add one to the z-coordinate, and normalize the result:

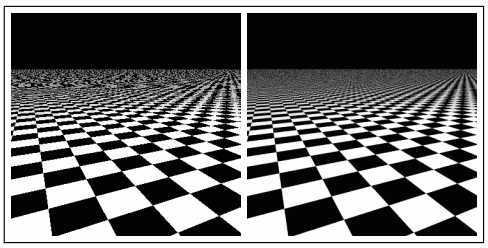
$$r = \frac{(d_x, d_y, d_z + 1)^T}{\|(d_x, d_y, d_z + 1)^T\|}$$

The texture coordinate corresponding to the direction d is then (r_x, r_y), i.e. we just omit the z-coordinate. Usually this mapping is performed by the graphics API, so that only the vertex normals have to be provided by the user and the respective texture coordinates are generated automatically.

Fortunately, generating the texture image is very easy: we just photograph a shiny sphere which we locate at the object's position in the scene. The picture of the sphere is the environment map (cf. Fig. 4.8c).

4.5.5 Texture Antialiasing

Texturing an object may cause an effect called *aliasing* when the frequency of the texture image does not match the resolution of the projected triangle's pixel resolution. On the one hand, it might happen that one screen pixel corresponds to many texture pixels (*minification*). On the other hand, many screen pixels may also map to one texel (*magnification*).



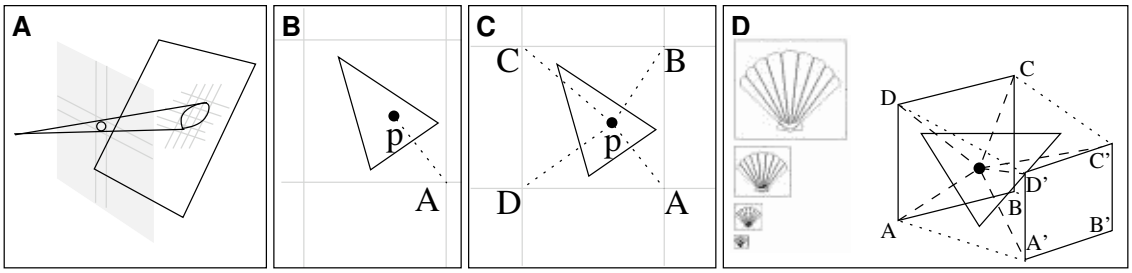


Figure 4.9: Aliasing: (a) pixel integration, (b) nearest neighbor, (c) bilinear interpolation, (d) mipmapping and trilinear interpolation.

We will discuss both effects in the next subsection and show solutions to these problems (the figure shows an alias-affected image on the left and the anti-aliased version on the right).

Magnification

Magnification corresponds to the case in which many screen pixels correspond to one texel only.

In a naive approach (*nearest neighbor*) we just use the color of the (integer coordinate) texel that is nearest to the (floating point) texture coordinate (cf. Fig. 4.9b). However, in the screen areas where magnification occurs, the texels will appear as large squares on the screen.

We can solve this by not looking at the texture as an area full of square-shaped color blocks, but instead as a grid of color samples. The pixel's texture coordinate lies in a rectangle determined by four texels. *Bilinearly interpolating* its color from the colors of these four neighbors in the grid results in better filtering (cf. Fig. 4.9c).

Minification

Let us consider the situation in which one screen pixel corresponds to a set of texels. Obviously, the screen pixel can take only one color.

In the *nearest neighbor* approach, we calculate the center of the backprojection, and lookup the nearest color in the texture. Using this color value unfortunately creates noise.

Similar to the magnification case we can also use *bilinear interpolation* to achieve some filtering effect. The problem is that just filtering over four texels may not be sufficient for strong minification.

The correct solution is to integrate over the colors of all texels which project to the current pixel. For simplicity assume a pixel with circular shape. Then the part of the 3D scene projecting into a pixel is a cone with apex at the camera position and opening through the pixel into the object space. Intersecting this cone with the triangle to be textured gives us an ellipse on this triangle. Transforming this object space ellipse into texture space yields another ellipse, that now contains all texels projecting onto the screen pixel in question (cf. Fig. 4.9a). Using a weighted sum of all texels inside this ellipse as the pixel's color results in an anti-aliased image.

Unfortunately, this is computationally very costly. There is another approximate solution which is easier to handle, called *Mipmapping*. We simply store the texture image in several pre-filtered resolutions (cf. Fig. 4.9d). For the texturing of a triangle the best texture resolution is chosen depending on the projected size of the triangle (usually, this is automatically done by the graphics API). Since mipmapping only chooses the right pre-filtered texture image, it can be used in conjunction with nearest neighbor or bilinear interpolation filtering.

In order to avoid popping effects when the system switches from one mipmap level to the next one, one can also use *trilinear interpolation* (cf. Fig. 4.9d). The color is looked up in the two best matching mipmap textures using bilinear interpolation for each of them. These two colors are then again linearly interpolated to smoothly blend between successive Mipmap levels.

Part II

Graphic APIs

In the previous chapter we have seen how to render 3D primitives, i.e. points, lines and polygons. This process involves very complex computations, hence implementing these algorithms in software would be very slow.

Fortunately, most parts of the local rendering pipeline are implemented in hardware on modern graphic cards. Exploiting this highly optimized special-purpose hardware greatly accelerates rendering. Modern graphic boards manage to render up to 20 million triangles per second.

In order to use the graphics hardware it is obviously not practical to access the hardware features directly, since this would result in code running on one specific graphic board only.

For that reason, low-level drivers encapsulate the specialties of the graphics hardware. The developer can then access the functionality on a higher level using *Graphic APIs*, resulting in portable code. There are two general types of graphic APIs, which can be distinguished by the level of abstraction they offer:

- *Render APIs* provide access to the graphic card's functionality through a set of rendering commands, e.g. for drawing triangles. Prominent examples are OpenGL and Direct3D.
- *Scenegraph APIs* work on a higher level of abstraction, using scene descriptions rather than a sequence of rendering commands. Prominent examples are OpenInventor and Java3D. Note that scenegraph APIs normally use rendering APIs for displaying geometry; for example, OpenInventor is build on top of OpenGL.

A major goal is code portability, i.e. we would like to write code which runs on a wide range of platforms, operating systems and graphic cards.

In the following two chapters we will introduce the OpenGL rendering API. In contrast to Microsoft's Direct3D, this API is supported on various platforms, like e.g. Unix, Windows and MacOS. We will first discuss the basic aspects of OpenGL, and then briefly mention advanced features. In the last chapter of this part we will present the underlying concepts of scenegraph APIs.

Note that we do not intend to give a detailed insight into either of the topics presented here. Refer to [Woo et al., 1999] for further information.

Chapter 5

OpenGL Basics

OpenGL pretty much implements the rendering pipeline the way we discussed it in the first part of the lecture. It is independent of the operating system (there are implementations for Windows, Unix/Linux and MacOS, for example), and it is supported by almost all graphic cards.

Since most parts of the rendering pipeline are nowadays implemented in hardware, OpenGL is a very efficient API for all kinds of graphics software. Even in the absence of hardware acceleration there are software implementations of OpenGL, and so OpenGL software can be used on these systems, too.

Actually, OpenGL is not just one library, but it is supplemented by several additional libraries, allowing access to OpenGL on several abstraction levels:

- The *GL Utilities* (GLU) contain utilities which can be used to simplify rendering, like e.g. functions for rendering spline surfaces (cf. Sec. 11.1.2) and quadrics (cf. Sec. 9.1.1).
- *Windowing APIs* are the interface to the underlying window system. Which library you need depends on your window manager: GLX is used for X-Windows, WGL for Windows and AGL for MacOS.
- *GUI APIs*, like the GL Utility Toolkit (GLUT) or Qt, further encapsulate the windowing APIs. They provide GUI elements (widgets) you can render into and a set of OpenGL wrappers for them. Note that these APIs are not part of OpenGL.

Our examples will be based on Qt (detailed reference can be found in [Dalheimer, 2002]). Besides many other features Qt provides the class `QGLWidget`, which implements a widget for OpenGL rendering. It contains three important methods which have to be overwritten:

- `initializeGL()`: Called once when the widget is initialized.
- `resizeGL()`: Called whenever the size of the widget changes.
- `paintGL()`: Called when the widget should re-draw its contents.

OpenGL was designed for C programming and hence does not support function overloading. Therefore, the name of each function is suffixed by the dimension of the coordinates and their scalar type. For example,

```
glVertex4fv (p)
```

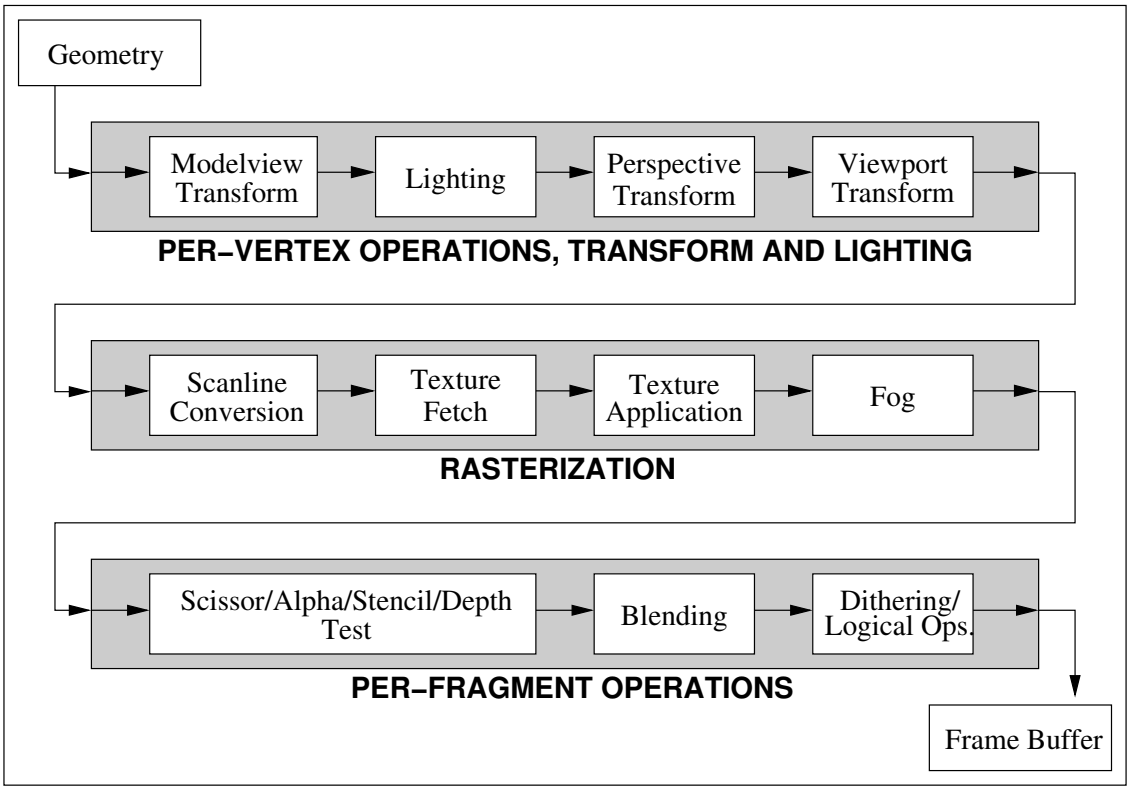


Figure 5.1: The OpenGL Rendering Pipeline.

corresponds to drawing a vertex (`glVertex`) in homogeneous coordinates (4D vector, indicated by `4`). The coordinates are specified by floating point scalars (`f`) and arranged in an array (`v`). Since the naming conventions are quite easy, we will not discuss them any further; note however that in the subsequent sections we will specify the function class only (e.g. `glVertex`) and omit the parameter specification.

The OpenGL pipeline is very similar to the rendering pipeline we discussed in the first part of the lecture, and consists of the following steps, which will be described in the remainder of this chapter (cf. Fig. 5.1):

1. Per-Vertex Operations (*Transform and Lighting*, T&L):
   (a) Modelview Transformation: transforms vertex positions from model coordinates to camera coordinates.
   (b) Lighting is computed for each vertex.
   (c) Perspective Transformation: applies the frustum map.
   (d) Viewport Transformation: scales to viewport dimensions.
2. Rasterization: in case of lines and polygons, this stage decides which pixels to fill and how to fill them.
   (a) Scanline Conversion: calculates the pixels spanned by a primitive.
   (b) Texture Fetch: fetches color values from texture images.
   (c) Texture Application: mixes texture color(s) and lighting color.
   (d) Fog: provides fog effects.

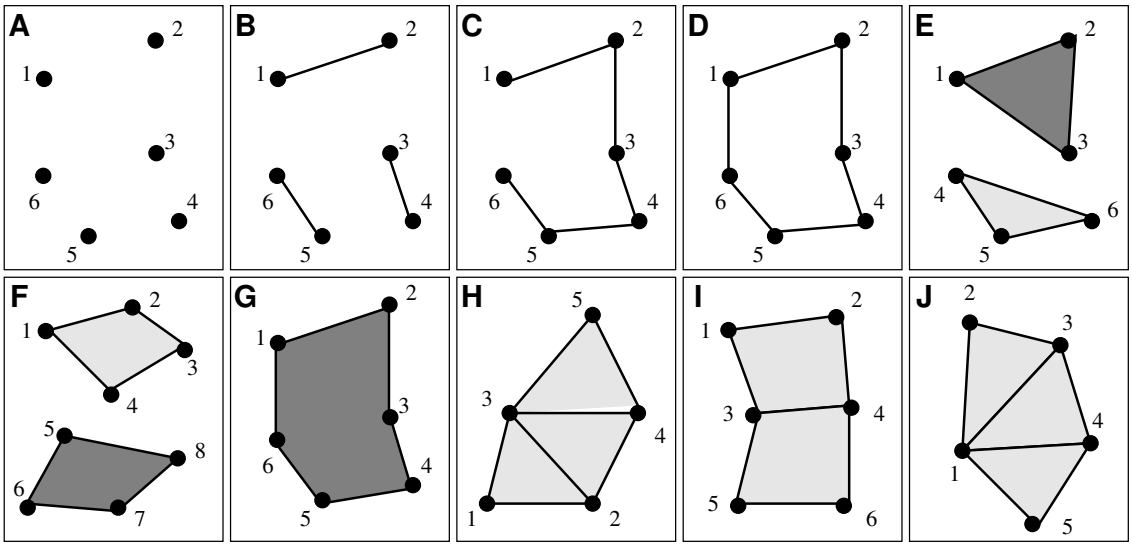


Figure 5.2: OpenGL Primitives: (a) `GL_POINTS`, (b) `GL_LINES`, (c) `GL_LINE_STRIP`, (d) `GL_LINE_LOOP`, (e) `GL_TRIANGLES`, (f) `GL_QUADS`, (g) `GL_POLYGON`, (h) `GL_TRIANGLE_STRIP`, (i) `GL_QUAD_STRIP`, (j) `GL_TRIANGLE_FAN`. Note that for face primitives the order of the vertices determines the front-facing and back-facing side. Front-facing faces are shaded light in the above figures, back-facing faces darker (assuming default CCW orientation).

3. Per-Fragment operations: for each pixel coming out of the previous stage, some further operations are performed to determine how to fill the pixel; in contrast to the rasterization stage, the operations are based on the pixel position on screen.
   (a) Scissor/Alpha/Stencil/Depth tests: these tests keep or discard pixels based on simple tests.
   (b) Blending: the pixel color can be mixed with the color in the frame buffer.
   (c) Dithering and Logical Operators (cf. Sec. 16.2).

5.1 Geometric 3D Primitives

OpenGL can process a large variety of primitives. All of them are specified by a set of vertices (cf. Fig. 5.2):

- `GL_POINTS`: Each vertex represents a single point.
- `GL_LINES`: Each pair of vertices represents a line segment.
- `GL_LINE_STRIP`: The vertices build a poly-line of connected segments.
- `GL_LINE_LOOP`: a closed line strip.
- `GL_TRIANGLES`: Each three vertices represent a triangle.
- `GL_QUADS`: Each four vertices represent a quadrilateral.
- `GL_POLYGON`: All vertices represent one single polygon.
- `GL_TRIANGLE_STRIP`: The first three vertices specify a triangle, and every other vertex corresponds to another triangle connected to the previous one.

Algorithm 4 Drawing a set of triangles in OpenGL.

```
glBegin(GL_TRIANGLES);
for (i=0; i<NUM_TRIANGLES; i++) {
  glColor3fv(my_colors[i]);
  glVertex3fv(my_vertices[i]);
}
glEnd();
```

- `GL_QUAD_STRIP`: Same method for quadrilaterals (note that a new quadrilateral is added using two additional vertices).
- `GL_TRIANGLE_FAN`: The first vertex is the center vertex, and all others build a set of connected triangles around it.

The vertices are specified in a block bounded by a `glBegin()` and `glEnd()`, as shown in Alg. 4. For each vertex we can additionally specify several properties (see next section); in the example code, we specified a color for each vertex.

5.2 State Machine

When drawing geometric primitives, we want to control how they appear on the screen. For example, we might give them specific material, lighting or shading parameters, or textures to be applied.

In order to avoid explicitly passing these parameters each time when specifying a vertex, OpenGL renders primitives according to its current *rendering state*. This state contains all rendering parameters, such as the ones listed above, and can be changed using a set of OpenGL commands. OpenGL is therefore said to implement a *state machine*.

This conceptual model exploits temporal locality: it is very probable that consecutive vertices share most of the rendering parameters. By using the state machine, we only need to inform OpenGL of changes in the parameters, instead of specifying all parameters for each vertex. Thereby, a lot of unnecessary data transfer is avoided.

The rendering state can be changed as follows:

1. Rendering parameters associated with vertices can be set using functions like `glColor()` (vertex color), `glNormal()` (normal vector) and `glTexCoord()` (texture coordinates).
2. Global rendering modes can be turned on/off using `glEnable(mode)` and `glDisable(mode)`; for example, to turn on lighting, use `glEnable(GL_LIGHTING)`.
3. Parameters for a specific rendering mode can be specified using functions depending on the mode. For example, lighting parameters can be set using `glLight(param,value)`.

Fig. 5.3 shows an example of what can be done by just changing the current rendering state. All images are based on the same scene. The first image was rendered without lighting, the second and third image were rendered with lighting using flat shading and Gouraud shading, respectively. In the bottom row, the objects were rendered using different material properties.

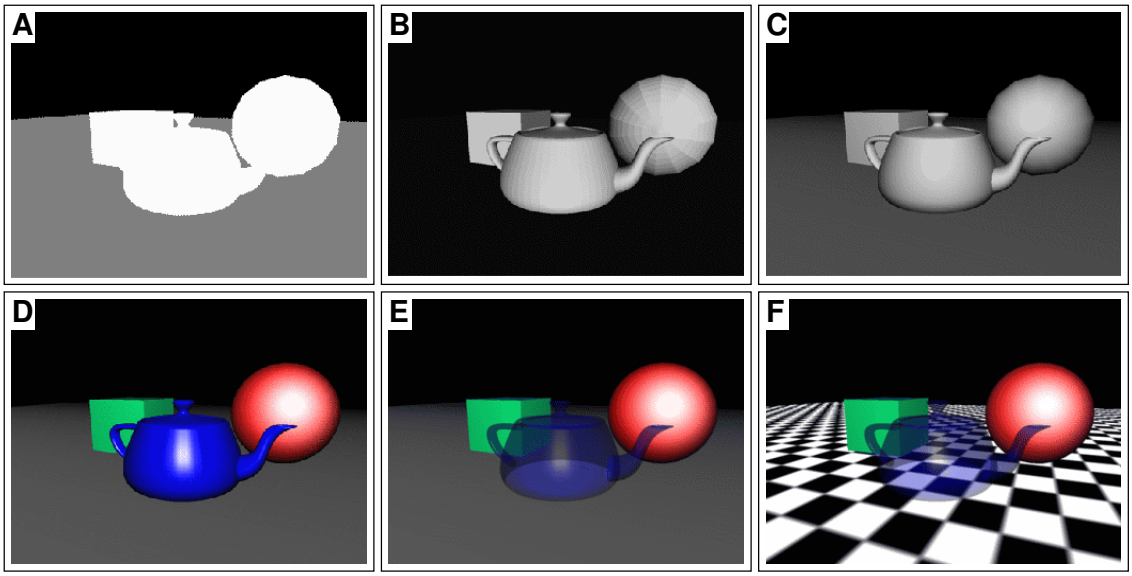


Figure 5.3: Effect of changing the rendering state: (a) no lighting, (b) flat shading, (c) Gouraud shading, (d) surface materials, (e) transparent teapot, (f) textured floor.

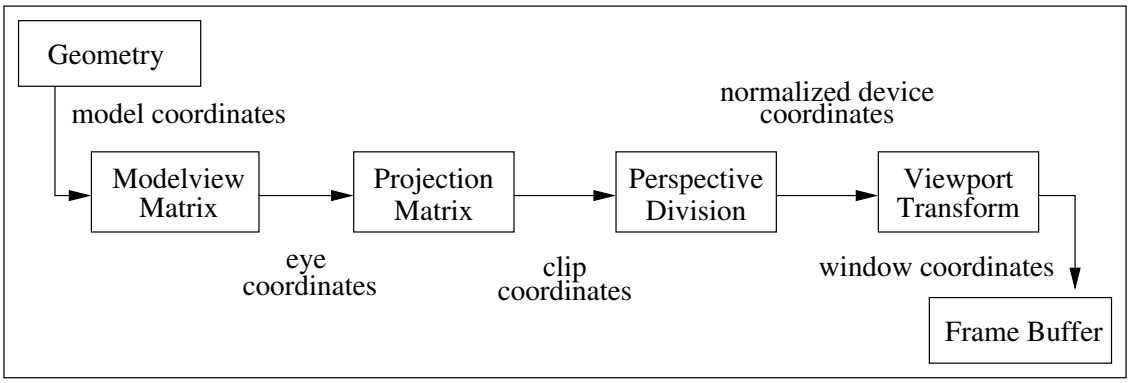


Figure 5.4: OpenGL Vertex Transformations.

5.3 Transformations

As discussed in the first part of the lecture, vertices pass through a set of transformations until they finally appear on the screen. In this section we will describe how to specify these transformations in OpenGL (cf. Fig. 5.4).

All matrices are specified using homogeneous coordinates, i.e. by 4×4 matrices. However, in OpenGL there are some details you must take into consideration, since they do not match mathematical intuition:

- The matrix elements are stored column major, i.e. a vector containing an OpenGL matrix contains the elements column by column.
- Transformations are post-multiplied, i.e. whenever you apply a new matrix operation, the corresponding matrix will be multiplied to the current matrix from right. This is unfortunately not what we want: successive transformations should be multiplied from left. Therefore, we need to do the matrix operations in inverse order.

5.3.1 Modelview Matrix

The *Modelview Matrix* is used to transform vertices specified in a local coordinate system (*model coordinates*) into the global coordinate system (*world coordinates*) and further into the coordinate system of the viewer (*camera/view coordinates*).

Changing this matrix requires the modelview matrix mode to be used:

```
glMatrixMode(GL_MODELVIEW);
```

Once this mode is set, the following operations on the modelview matrix are available:

- `glLoadIdentity()`: sets the matrix to the identity matrix.
- `glLoadMatrix()`: loads a matrix specified by a vector of 16 elements in column-major order.
- `glMultMatrix()`: multiplying a matrix from right to the current matrix.

The above functions suffice to specify any transformation. However, we would need to construct a transformation matrix, and multiply it to the current modelview matrix each time. Therefore OpenGL offers the following functions for convenience:

- `glRotate()`: rotation specified by axis and angle.
- `glScale()`: scaling by x-, y- and z-scaling factors.
- `glTranslate()`: translation by a given vector.
- `gluLookAt()`: sets up the camera system, given in terms of the viewer's position, the center of the scene and the up-vector.

5.3.2 Projection Matrix

The *Projection Matrix* transforms the scene from eye coordinates into the homogeneous cube $[-w; w]^3$ (*clip coordinates*). Which part of the scene is visible depends on the type of projection (perspective or orthogonal) and its parameters.

Similar to the modelview matrix, the projection matrix mode must be activated first:

```
glMatrixMode(GL_PROJECTION);
```

Analogously to the modelview matrix operations, we can perform basic operations with the projection matrix using the functions `glLoadIdentity()`, `glLoadMatrix()` and `glMultMatrix()`. Again, functions are offered for convenience:

- Perspective projection. If the frustum is symmetric you can use `gluPerspective()`; for the general unsymmetric case you can use `glFrustrum()`.
- Orthogonal projection, using `glOrtho()` for the general 3D case and `gluOrtho2D()` for 2D drawings.

Algorithm 5 A small example illustrating OpenGL transformations.

```
void GLWidget::resizeGL(int w, int h) {
  glViewport(0,0,w,h);                              /* set new viewport */
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();                                 /* set new frustum */
  gluPerspective(45,(float)w/(float)h,0.1,10.0);
  glMatrixMode(GL_MODELVIEW);                       /* back to default mode */
}

void GLWidget::paintGL() {
  glLoadIdentity();                                 /* reset modelview-m. */
  gluLookAt(...);                                   /* specify viewer */
  glTranslatef(center.x, center.y, center.z);       /* Rotate the scene */
  glRotatef(angle, axis.x, axis.y, axis.z);         /*   around center */
  glTranslatef(-center.x, -center.y, -center.z);
  draw_my_geometry();                               /* draw primitives */
}
```

5.3.3 Viewport Transformation

Once we have the objects in clip coordinates $[-w, w]^3$, we perform *perspective division* by dehomogenization, resulting in *normalized device coordinates* in $[0, 1]^3$. Then, in a last step, we scale these coordinates according to the size of the *viewport*, yielding pixels in *window coordinates* $[0, width - 1] \times [0, height - 1] \times [0, 1]$.

For these transformations only the definition of the viewport's size is required, which can be set using `glViewport()`.

Alg. 5 shows a small example.

5.3.4 Matrix Stack

OpenGL also maintains a *matrix stack* for modelview and projection matrices. The former has at least 32 entries, the latter at least two. A copy of the current matrix can be pushed on the stack using `glPushMatrix()`, the topmost matrix can be popped from the stack by `glPopMatrix()`.

The modelview matrix stack can be useful for designing hierarchical objects, like e.g. a robot's arm. The robot's forearm is mounted on its upper arm, which in turn is attached to the robot's shoulder. The relative position of these parts can be described by transformation matrices.

In order to draw the robot, its model-to-world matrix sets up position and orientation, such that the body can be drawn. In order to draw the forearm we have to multiply this matrix by the matrix representing the shoulder angle and the matrix corresponding to the elbow angle.

To get back in this hierarchy of local coordinate systems we would have to multiply by the inverse matrices. Using the matrix stack this can be implemented much easier by pushing (storing) and popping (restoring) matrices.

5.4 Lighting

OpenGL implements the Phong-Blinn lighting model (cf. Sec. 2.3.1), i.e. light is assumed to consist of ambient, diffuse and specular components. The intensity/reflectivity of these three components

must be specified for both the light sources and the objects' materials. In addition to that, OpenGL supports more features for light sources:

- Local light sources as opposed to directional light sources.
- A spotlight factor (cf. Sec. 2.3.2) controlling how much the light intensity decreases with growing angle to the main emission direction.
- A light attenuation factor (cf. Sec. 2.3.2) controlling how much the light energy attenuates with growing distance from the light source.
- Depth cueing (cf. Sec. 2.3.2), i.e. light can change its color spectrum according to atmospheric effects.

OpenGL supports at least eight light sources in the scene, and their parameters can be set using `glLight()`. Lighting and the light sources to be used must be activated:

```
glEnable(GL_LIGHTING); glEnable(GL_LIGHT0); ...; glEnable(GL_LIGHT7);
```

Surface material properties can be specified using the function `glMaterial()`. More specifically, one can adjust the ambient, diffuse and specular reflection factors, the shininess factor, and the light intensity emitted by the primitive itself.

Light intensities are computed per vertex, not per pixel, based on the surface material and light source properties. Lines and polygons are then filled using these intensities, either using flat shading or Gouraud shading (cf. Sec. 4.4). The shading model can be chosen by `glShadeModel()` with parameter `GL_FLAT` or `GL_SMOOTH`, respectively.

5.5 Texturing

A very important technique for increasing the visual detail of objects is texturing (cf. Sec. 4.5). Note that we will only discuss simple 2D texturing here; more advanced texturing techniques will be considered in the next chapter.

5.5.1 Creating Texture Objects

Before we can use a texture we need a way to identify it. In OpenGL this is done using texture objects, which can be created using `glGenTextures()` and activated by calling `glBindTexture()`.

Afterwards we can assign a texture image to the object using `glTexImage2D()`. The texture image is a 2D array of color values. When assigning the texture buffer we must also tell OpenGL how to interpret the color values in the buffer (e.g. RGB, RGBA, etc.), and how these values should be stored in the graphic card memory.

Note that for performance reasons OpenGL supports texture sizes being powers of two only. Hence, you might need to rescale your texture first. Also, OpenGL does not support any image formats (like GIF or JPG). Note that Qt, however, provides functionality to load, save and scale images.

Algorithm 6 A texturing example in OpenGL.

```
void GLWidget::initializeGL() {
  glGenTextures(1, &texid_);
  glBindTexture(GL_TEXTURE_2D, texid_);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
  glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
  glTexImage2D(GL_TEXTURE_2D,0,GL_RGB,w,h,0,GL_RGBA,GL_UNSIGNED_BYTE,pixels);
  glEnable(GL_TEXTURE_2D);
}

void GLWidget::paintGL() {
  ...
  glTexCoord2fv(my_tex_coord);
  glVertex3fv(my_vertex);
  ...
}
```

5.5.2 Texture Fetch and Texture Application

The texture lookup is based on the texture coordinates specified for the vertices of the primitive (`glTexCoord()`). When specifying coordinates, $(0,0)^T$ refers to the lower left corner of the texture and $(1,1)^T$ to its upper right corner. In case larger values are specified, OpenGL can clamp or repeat the texture at its boundaries (`glTexParameter()`).

As already discussed in Sec. 4.5, simply backmapping the screen pixel might introduce aliasing effects due to minification and magnification. Therefore, OpenGL supports a variety of antialiasing techniques, e.g. nearest neighbor interpolation, bi-linear interpolation or tri-linear interpolation using mipmaps. The filtering technique to be used can be selected using `glTexParameter()`.

After the corresponding texels have been fetched from the texture image, the pixel colors have to be computed. OpenGL supports several combinations of the texture color and the color computed by the lighting model, to be set by `glTexEnv()`. Alg. 6 shows some example code.

5.6 Fragment Operations

In the last step of the OpenGL pipeline, several operations can be performed on fragment level. A *fragment* is a part of a primitive occupying one pixel; therefore, it is defined by its 2D pixel position, its depth (i.e. the distance between the corresponding point in object space and the image plane) and its color (as computed using lighting and textures). Not yet mentioned, the color actually contains a fourth component, the α-*value*, specifying the opacity of the pixel.

First, some tests are performed which either discard or keep the fragment; if all tests succeed, the fragment is rendered. The pixel color does not depend on the fragment's color only, but instead might also depend on the color currently in the frame buffer. Combining both values is referred to as *blending*. We will discuss both steps in the subsequent sections. Note that we will not discuss dithering and logical operations here.

Figure 5.5: Alpha Test: (a) coarse leaf geometry, (b) leaf texture, (c) application of the texture using the alpha test, (d) resulting tree.

5.6.1 Fragment Tests

In some cases, not all of the incoming fragments should be rendered. Therefore, they are fed into a series of tests, and if any of them fails, the fragment is discarded. As we will see, these tests can be used to implement masking and occlusion. Note, however, that all tests are turned off by default and need to be activated first.

Scissor Test

The scissor test can restrict rendering to a specific screen space rectangle. If the fragment is outside this area, it will be discarded.

`glEnable(GL_SCISSOR_TEST)` enables the scissor test, and the rectangle can be specified using `glScissor()`.

Alpha Test

The alpha-component of the fragment can be tested against a given reference value using comparison operators ($<, \leq, >, \geq, =, \neq$). If the test fails, the fragment is discarded.

The Alpha Test can be used, e.g., to discard fully transparent pixels of a primitive. Using this technique one can use a coarse geometry approximation and put the shape's details into a partially transparent texture. Consider a tree, for example. Representing it by its actual geometry, i.e. modeling each leaf by many polygons, would be too costly. Therefore, one would approximate the geometry of the leaves by very few polygons only, textured using an image of a leaf. However, a leaf does not have rectangular shape, so most of the texture would actually be transparent (i.e. have a zero alpha value). We can discard all those pixels using the alpha test, keeping only those pixels belonging to the leaf. Fig. 5.5 illustrates this.

Once being enabled by `glEnable(GL_ALPHA_TEST)`, both the reference value and the comparison operator can be set with `glAlphaFunc()`.

Stencil Test

OpenGL provides another buffer, called *stencil buffer*, that has an integer entry for each framebuffer pixel. When rendering a fragment the stencil test compares the corresponding entry in the stencil buffer against a reference value and keeps or discards the fragment according to the result of this test. In case the test succeeds, the stencil buffer entry can be changed before passing the fragment over to the next fragment test. Available operations are setting the buffer entry to zero or to the reference value, increasing or decreasing it, or inverting the current value bitwise.

One application for stencil tests is the use of masks. Consider a flight simulator, for example. A part of the screen is occupied by the cockpit, and we do not want the terrain to overwrite this part. We can achieve this in three steps:

1. Clear the stencil buffer, i.e. set all values to zero.
2. Draw the cockpit image. While doing so, configure the stencil test to always pass, and the stencil function to increase the stencil buffer entry.
3. Draw the terrain. This time, however, the stencil test allows fragments with a zero stencil value only, and the stencil function does not modify the stencil buffer.

The stencil test can be activated calling `glEnable(GL_STENCIL_TEST)`. `glClear()` clears the buffer; the value to be used for that can be set using `glClearStencil()`. The functions `glStencilFunc()`, `glStencilMask()` and `glStencilOp()` allow for a configuration of the test.

Depth Test

OpenGL manages yet another buffer, the so-called *Z-buffer*, which contains the depth value of the fragments. In a last test, the current fragment's depth value is compared against the corresponding entry in the Z-buffer and discarded if the test fails.

The Z-buffer can be used to resolve occlusion and visibility by discarding all fragments for which the Z-buffer contains smaller values. This topic is further discussed in Sec. 12.2.2.

The depth test can be enabled by the function call `glEnable(GL_DEPTH_TEST)`. Similar to the stencil buffer, `glClear()` and `glClearDepth()` can be used to reset the buffer to arbitrary values, and `glDepthFunc()` configures its functionality.

5.6.2 Blending

In some cases, objects might not be fully opaque. Consider colored glass for example; what you actually see is not just the color of the glass, but also its background shining through it.

To achieve such effects, a fourth component is added to the color: the α-*value*. This value specifies the opacity of the object; zero thus corresponds to fully transparent material, whereas one corresponds to fully opaque material. You can set these values using `glColor()` or `glMaterialfv()`.

When it comes to rendering translucent fragments, the color of this fragment is combined with the color found at the corresponding position in the frame buffer, according to the following formula:

$$rgba_{dest} \leftarrow f_{src} \cdot rgba_{src} + f_{dest} \cdot rgba_{dest}$$

In this formula, $rgba_{src}$ is the color of the current fragment, and $rgba_{dest}$ is the color found in the frame buffer. The weighting coefficients f_{src} and f_{dest} can be chosen using `glBlendFunc()`. This function takes two parameters, one for the source and one for the destination component. For example, one can specify `(GL_ONE,GL_ZERO)`, resulting in the current fragment overwriting the frame buffer, or `(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA)`, causing to blend both values using the alpha value of the source fragment. The latter is what you normally might want to achieve when using blending.

Translucent objects cause problems with the Z-buffer. Note that objects need to be rendered now although other objects (translucent ones) occlude them; thus we cannot simply discard fragments with a larger depth value than the current one. One solves the problem usually using two rendering passes; first, all opaque objects are rendered, and then all translucent objects are rendered in back-to-front order, this time however with the Z-buffer being read-only.

Blending can be enabled using `glEnable(GL_BLEND)`.

Chapter 6

Advanced OpenGL

In the previous chapter we discussed how 3D primitives can be rendered using OpenGL. However, OpenGL offers much more functionality, allowing for more advanced rendering effects or even better performance.

In the subsequent sections we will briefly describe what else OpenGL is capable of. However, we will not give a detailed explanation here; refer to [Woo et al., 1999] for further information on this topic.

We will also briefly discuss the OpenGL extension mechanism; again you are referred to other sources for more information.

6.1 Imaging Subset of OpenGL

Although OpenGL is normally used in the context of 3D rendering, it also provides functions which can be used for 2D image processing.

It distinguishes two types of pixel primitives: *bitmaps* and *images*. Bitmaps are 2D arrays of bit-masks, which can be used for pattern filling or displaying font characters. In contrast to that, images are 2D arrays of RGB (or RGBA) colored pixels. In the following, we will focus on images.

Note that OpenGL does not support image formats, such as PNG, GIF or JPEG, and it does neither provide functions to load or save image files. Fortunately, Qt does provide various imaging functions, so you might want to use these to complement the function set of OpenGL.

The position for drawing images in OpenGL is controlled by one vertex, the *raster position*. This position gets transformed in the usual way, i.e. by modelview and projection matrix, and if this vertex is not clipped, the image will be drawn using the corresponding pixel position as its lower left corner.

Images can also be retrieved from graphic card memory: OpenGL supports saving the contents of various buffers (frame buffer, Z-buffer, etc.) into an array of pixels. This functionality can be important for algorithms requiring multiple rendering passes, since partial results can be temporarily stored.

Finally, the imaging functionality contains transfer, mapping and zooming functions; the RGBA channels can be rescaled and biased, the color depth can be changed, colors can be mapped to other ones, and pixels can be magnified, reduced or flipped.

Some more functions are part of the `ARB_imaging` extension (cf. Sec. 6.6), which has been part of the OpenGL core since version 1.2. This subset provides functions for image convolution filters and histograms (cf. Sec. 15.2 and 15.5). Furthermore, it adds a color matrix stack with at least two entries, providing easy conversions between different color spaces.

6.2 Advanced Texturing

As already hinted at in Sec. 5.5, besides 2D textures, 1D and 3D textures are supported as well. Note that their size is restricted to be powers of two in each dimension.

OpenGL provides methods for automatic texture coordinate generation. For example, it supports the calculation of texture coordinates for spherical environment mapping (cf. Sec. 4.5.4). Furthermore, it provides projective textures: similar to projection matrices, they are controlled by a 4×4 texture matrix transforming the 4D homogeneous texture coordinates.

Another important supported feature is *Multitexturing*. Modern graphic cards have several texture units, allowing for multiple textures to be applied in one pass. Note that multitexturing is quite common: in a racing simulation, e.g., a car might not only be supplied with a material texture, but also with environment maps simulating reflections.

6.3 Interaction

In some applications it is important to interact with the objects in the scene. Therefore we need a way to find out on which object in the scene the user clicked. OpenGL offers two different ways to solve the problem.

One way is to calculate the intersection of the ray from the eye through the selected pixel with all objects in the scene. OpenGL provides a function to calculate this object space ray (`gluUnproject()`). Note that the required ray intersection has to be computed independently from OpenGL. We will see in section 14 that this can be quite expensive and how it can be accelerated.

OpenGL also offers the more convenient *picking mode*; in this mode, each object is associated with an identifier during rendering. When the user clicks on a position, the list of objects projecting onto the clicking position can be retrieved by one special rendering pass.

6.4 Advanced Primitives

As we will see in the next part of the lecture, there are much more geometry representations than just polygonal meshes. OpenGL can handle polygons only. However, other representations can be used to describe objects, which are then converted into polygonal meshes by OpenGL during rendering.

The GL Utilities library (GLU) provides functions for drawing quadrics, i.e. primitives defined by quadratic functions, such as spheres, cylinders and two-dimensional discs (cf. Sec. 9.1.1). Furthermore, the GL Utility Toolkit (GLUT) offers functions for rendering Platonic Solids (cf. Sec. 8.3), spheres, cones and tori. Also, the famous Utah teapot can be drawn.

Besides these, OpenGL can display higher order polynomial curves and surfaces. These surface representations will be discussed in Sec. 11.1.1 and Sec. 11.1.2.

6.5 Performance Improvement

With the functions we have discussed in the last chapter we can render polygonal meshes. However, rendering can be done much faster. Note that although vertices are shared by several triangles, we send them to the graphic hardware every time they are referenced by a triangle (see Alg. 4). Even

worse, we also send their texture coordinates and normals multiple times. This kind of rendering is referred to as *immediate mode*, and leads to unnecessary data transfer and T&L operations.

OpenGL provides some mechanisms to reduce this overhead drastically, which we will discuss now.

6.5.1 Display Lists

A series of OpenGL commands can be merged together into a *Display List* by enclosing them within special OpenGL calls. Then, all these commands can be executed by rendering the display list using one function call only.

In contrast to immediate mode rendering, OpenGL optimizes the commands in a display list in order to minimize data transfer and function calls; you can look at this as some kind of compiling process.

Also, display lists can be stored in the graphics card memory, thereby avoiding data transfer. The main drawback of display lists is the fact that they are read-only: Once they are created, they cannot be changed anymore. Hence, if a change becomes necessary (e.g. in dynamic scenes), we need to re-compile the whole display list.

6.5.2 Vertex Arrays

Using Vertex Arrays, all vertex coordinates, normals and texture coordinates are put into separate arrays. The vertices and their associated data can then be referred to by an index into these arrays. Since a triangle is specified by three indices, a whole mesh can be rendered by one function call providing an array containing all its triangle indices.

The advantage of this method is that it enables the use of the so-called *Post-T&L-cache* on the graphics card. This cache stores the last T&L results (typically about 16). When a vertex to be rendered is found in this cache, the cached data can be used, instead of computing T&L again.

6.6 OpenGL Extensions

One of OpenGL's strengths is also its main drawback. Since a lot of organizations (the so-called Architecture Review Board, *ARB*) are involved in the further development and extension of OpenGL, its response time to the latest features of modern graphic cards is rather long.

Therefore, vendors can propose *OpenGL extensions* that offer additional functionality. Since only few companies are involved in designing and implementing such libraries (typically the graphic card vendors), new hardware functionality gets accessible much sooner.

To avoid naming conflicts, extensions have a particular prefix or suffix. Examples are `NV` (NVidia extensions), `ATI` (ATI extensions), `MESA` (Mesa extensions) and `SGI` (SGI extensions). If a group of vendors designs an extension, `EXT` is used, and finally, if the ARB decided that an extension will most probably get part of the next OpenGL specification, it is prefixed with `ARB`. The extension itself and its constants and functions are tagged with this identifier, for example:

Extension	:	`GL_`*`EXT`*`_bgra`
Extension routine	:	`glMultiTexCoord2f`*`ARB`*
Constants	:	`GL_REGISTER_COMBINERS_`*`NV`*

The extensions are maintained in a registry (see [Silicon Graphics, 2003]) in order to ensure that OpenGL code can be compiled everywhere, regardless of whether or not an extension is available.

Chapter 7

Scenegraph APIs

In contrast to rendering APIs, like e.g. OpenGL, *Scenegraph APIs* provide a more abstract access to 3D graphics. Instead of specifying the geometry on the level of 3D primitives, a scene description is supplied.

Scenegraph APIs usually do not implement rendering functionality on their own. Instead, they are based on rendering APIs, i.e. they basically convert the high-level scene description to the low-level primitives and instructions provided by rendering APIs.

In this chapter we will briefly discuss OpenInventor [Wernecke, 1994], a commonly used scenegraph API developed by SGI that is based on OpenGL.

Scenegraph

The scene is represented as a *directed acyclic graph* (DAG) that builds a hierarchical scene description. Each node in the graph represents one property of the scene. The basic node types include the following:

- geometry nodes, representing the geometry of an object.
- Material nodes, storing a set of material properties.
- Transformation nodes, storing a 3D transformation.
- Texture nodes, referencing a texture.
- Light nodes, giving the parameters of a light source in the scene.

The properties described by a node are used by its child-subtrees in the graph. For example, a material node specifying red color will result in all its children being rendered using this color.

Due to the representation of the scene by a DAG, cross references within the tree structure are allowed. That is, a given node can be referenced by a set of other nodes, as long as no cyclic path is constructed. For example, in a forest you might have many trees, but most of them share the same geometry. Therefore, you would create geometry nodes for the different tree types, and then reference them multiple times by nodes specifying their positions.

Supported Geometry Primitives

The geometric primitives are basically those provided by OpenGL. Besides the GLUT primitives (spheres, cylinders, cones, boxes etc.) and NURBS (cf. Sec. 11.1.2), OpenInventor also provides polygonal meshes in form of face sets (i.e. a set of triangles) or indexed face sets (i.e. triangles reference vertices by indices) and triangle strips. However, new nodes can be defined if needed.

Scenegraph Actions

Rendering a scene mainly consists of traversing the graph and calling a draw-method for each node. This kind of behavior is implemented in terms of *actions*.

Using a *render action* each node calls its special rendering commands during the depth-first traversal of the scene graph. Material nodes may e.g. change the current material (i.e. change the rendering state), while geometry nodes will send the necessary rendering commands to draw their primitives.

Similar to render actions there are also *pick actions* that use ray intersections or the OpenGL picking mechanism to find the object the user clicked on.

Advanced Nodes

OpenInventor defines some special nodes, some of which we will briefly mention here:

- Shape hints, specifying the quality level to be achieved when rendering the sub nodes.
- Level-of-detail nodes, providing a hierarchy of surfaces to be used for different levels of detail. Using them a coarse geometry proxy is rendered if the object is far away, changing to meshes containing more and more details when the viewer approaches the object.
- Manipulators, allowing interactive transformations of objects. For example, one could assign a manipulator to the object, which causes the object to transform when the manipulator is dragged. This gives the user a much easier access to interactive scenes than offered by the selection mode of pure OpenGL only.

Part III

Geometry

In the last parts of the lecture we described how to render a scene into an image using local illumination methods. We assumed that the scene is represented as a set of points, lines and polygons.

In fact we can approximate the surface of arbitrary 3D objects by polygonal meshes very well, hence a rendering pipeline built for polygons is indeed useful. We will discuss several properties of such meshes, and present data structures for compact storage as well as efficient traversal.

However, there are also other representations for 3D objects. One of them is an approach called *Constructive Solid Geometry* (*CSG*). CSG objects are based on simple geometric objects, such as spheres. Complex CSG models can be generated by combining several simpler objects using boolean operations and transformations. They are specified in terms of implicit functions describing the volume they occupy in 3D space.

Note that implicit functions allow for checking at each point in 3D space whether it is inside the object's volume or not. Objects defined in such a way are therefore also referred to as *volumetric objects*. One way to visualize such volumetric objects is to extract iso-surfaces from the volume, represent them as meshes, and render them using the rendering pipeline. However, there are also methods that directly render volumetric data without creating a mesh representation first. We will discuss both techniques in this part.

Another parametric geometry representation are *Bezier*- and *spline* curves or surfaces. These objects are modelled by polynomials of higher degree such that normal vector or curvatures are well defined. This is important if such objects are to be connected to each other in a smooth manner, like e.g. in CAD applications. These surfaces are defined by a set of control points, such that the surface can intuitively be edited by moving these points in space, making them suitable, e.g., for CAD applications.

Note that this chapter is not intended to discuss all the topics in detail; this is deferred to the lectures ,,Computer Graphics II“ and ,,Geometric Modeling I+II“.

Chapter 8

Polygonal Meshes

The surface of 3D objects can be represented (or approximated) by a set of polygons. These polygonal faces are placed next to each other such that they share common edges and vertices with their neighboring faces. Therefore this representation is referred to as a *polygonal mesh*.

A polygonal mesh consists of *geometry* as well as *topology*. The first one is specified by the vertices (corner points) of the mesh and defines the shape of the object, whereas the latter one is defined by the edges and faces of the mesh, thereby specifying the neighborhood relationship between the vertices.

However, the geometry of a triangle mesh is not really defined by its vertices alone, but by the piecewise linear surface spanned by these vertices and the triangles. In a general polygonal mesh we might have polygons consisting of more than three vertices. Then, although we know the polygon's edges, it is not clear how to define its face since it will not be planar in general. One pulls back from this ambiguity by neglecting edges and faces when referring to geometry.

Note that for the rendering process this ambiguity is resolved by splitting general polygons into triangles: triangular faces are uniquely determined by their vertices and edges.

8.1 Topological Mesh Properties

8.1.1 Duality

For a polygonal mesh one can define its so-called *dual mesh* (in this context, the original mesh is called the *primal mesh*). In the dual mesh any k-dimensional entity of the primal mesh is replaced by a $(2-k)$-dimensional entity. Specifically, any face is replaced by a vertex, vertices are transformed into faces, and only edges keep their dimension.

The above rules specify the topology of the dual mesh only. In order to define its geometry one places the vertex corresponding to a face into the barycenter of that face, and connects two vertices by an edge, if the corresponding faces are adjacent in the primal mesh. The face a vertex is transformed into is then defined by the edges in the dual mesh surrounding that vertex. Fig. 8.1a and 8.1b show an example.

8.1.2 Two-Manifolds and Closedness

A very important property of polygonal meshes is *two-manifoldness*. Intuitively, a mesh is called two-manifold if it represents a simple surface without additional parts being attached to it. Using a more formal definition, a mesh is two-manifold if it has the following three properties:

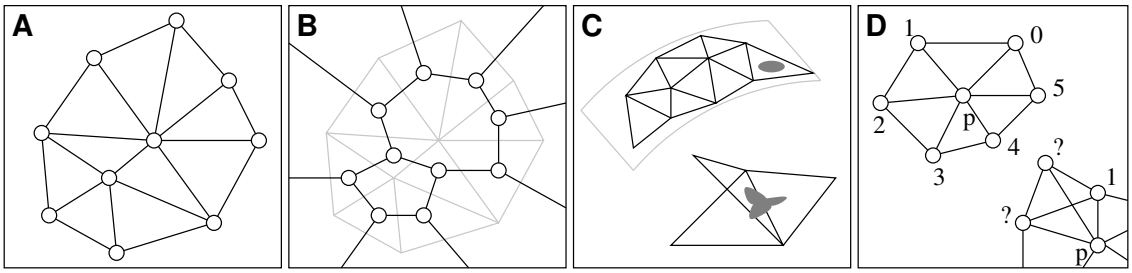


Figure 8.1: Mesh properties: (a)(b) primal and dual mesh, (c) local disc property, (d) edge ordering property.

- *Local Disc Property:* On a two-manifold mesh one can always find a ε-ball around any point on the mesh (i.e. a sphere of radius $\varepsilon > 0$ around that point) such that the intersection of that ε-ball with the mesh is homeomorphic to a planar disc. In the case of points on the boundary, this intersection can be homeomorphic to a half-disc (cf. Fig. 8.1c).

- *Edge Ordering Property:* If the mesh is two-manifold, one can order the neighboring vertices of a given vertex uniquely in a clockwise or counter-clockwise circular manner (cf. Fig. 8.1d). *Complex vertices* are vertices for which this ordering cannot be constructed.

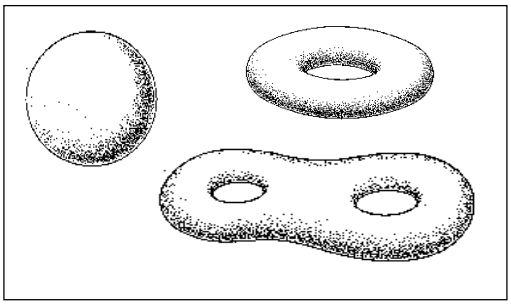
- *Face Count Property:* Each edge of a two-manifold mesh has exactly two adjacent faces if it is an interior edge and exactly one adjacent face if it is a boundary edge (cf. Fig. 8.1c). An edge having more than two (or one) adjacent faces is called a *complex edge*.

Another important property of polygonal meshes is *closedness*: A mesh is called closed, if it has no boundary (and therefore, any edge in the mesh has exactly two adjacent faces if the mesh is also manifold).

8.1.3 Euler Formula

The famous Euler formula [Coxeter, 1969] provides a correspondence between the number of the different entities (vertices, edges, faces) in a polygonal mesh:

$$V - E + F = 2(1 - g)$$



In this formula...

- V, E and F correspond to the number of vertices, edges and faces in the mesh, respectively.

- g is the *genus* of the object. The genus is the number of handles in an object. For example, a sphere has genus zero, a torus has genus one, and a double-torus has genus 2 (see above figure). Of course, given a mesh, one can derive the genus just according to the Euler formula:

$$g := 1 - \frac{V - E + F}{2}$$

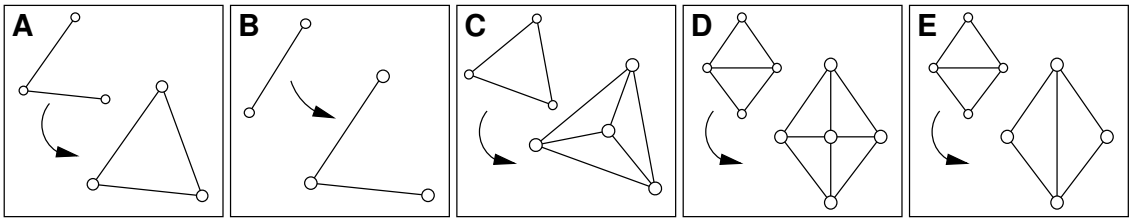


Figure 8.2: Euler Proof: (a)(b) adding edges, (c) face split, (d) edge split, (e) edge flip.

Proof: We will prove the Euler formula for the case of triangle meshes with genus zero, i.e.

$$V - E + F = 2.$$

Let us consider the planar case first. We can show the validity of the formula by induction. We have to show that the formula holds for the simplest planar object: a single vertex. Note that since the formula only holds for closed meshes we have to add one face representing the empty space around the mesh. Thus, we get

$$V - E + F = 1 - 0 + 1 = 2.$$

Now, assume a mesh for which the Euler formula holds. We need to show that by extending the mesh, the formula still remains valid. This is trivially equivalent to showing

$$\Delta V - \Delta E + \Delta F = 0,$$

where ΔV, ΔE and ΔF represent the change in the number of new vertices, edges and faces, respectively. It can be shown that any planar graph can be built from scratch by successively adding new edges. New vertices must be connected to the mesh, and new faces can be created by adding several edges. There are two cases to consider:

- In case we insert an edge between two existing vertices (cf. Fig. 8.2a), we generate a face (this is implied by the fact that the mesh had only one component before):

$$\Delta V - \Delta E + \Delta F = 0 - 1 + 1 = 0$$

- If we attach an edge at some vertex (cf. Fig. 8.2b), we will also have to add one new vertex to the mesh in order to have an endpoint for the new edge:

$$\Delta V - \Delta E + \Delta F = 1 - 1 + 0 = 0$$

Having shown that extending the mesh does not falsify the formula we can conclude by induction that the Euler formula holds for planar triangle meshes of genus zero.

The 3D case can be shown similarly. First consider the simplest three-dimensional closed triangle mesh, a tetrahedron. Counting the vertices, edges and faces we get

$$V - E + F = 4 - 6 + 4 = 2.$$

Assuming we have a closed 3D mesh for which $V - E + F = 2$ holds, we can check what happens with the left hand side of the formula when extending it:

- *Face Split* (cf. Fig. 8.2c): Adding a new vertex into a face requires the insertion of three edges as well, connecting the new vertex to the face's vertices. This increases the number of faces by two (we split one triangle into three). In summary we get

$$\Delta V - \Delta E + \Delta F = 1 - 3 + 2 = 0$$

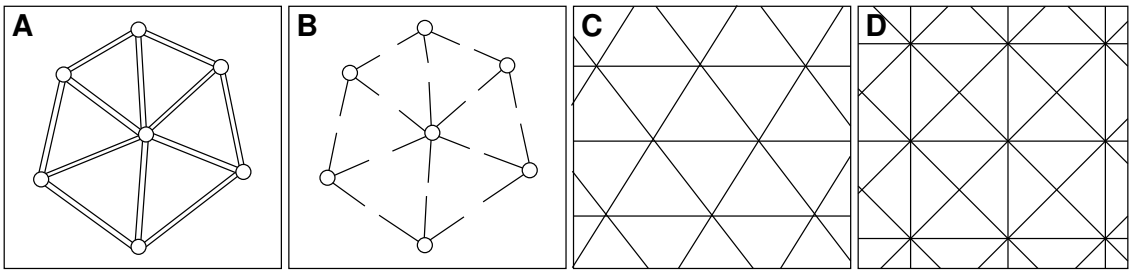


Figure 8.3: Properties of Meshes: (a)(b) halfedges, (c)(d) regular meshes.

- *Edge Split* (cf. Fig. 8.2d): When inserting a new vertex on an existing edge we need to split the adjacent faces by adding edges between the new vertex and the vertices on the opposite sides of the faces. Hence, we get

$$\Delta V - \Delta E + \Delta F = 1 - 3 + 2 = 0$$

Again we have shown by induction that the formula remains valid when we insert new vertices using the above operators. Every closed genus zero triangle mesh can be constructed starting from a tetrahedron by the operators edge-split, face-split and edge-flip (cf. Fig. 8.2e). The first two have been shown to respect the Euler characteristic, the latter one obviously also does not change it. Therefore we have shown that the Euler formula holds for all closed triangle meshes with genus zero. □

8.2 Triangle Meshes

When restricting to pure triangle meshes we can derive more interesting facts about the correspondence between the numbers of different entities from the Euler formula. For a given triangle mesh the Euler formula states that

$$V - E + F = 2(1 - g) = c,$$

where c is a constant usually being small compared to V, E and F.

Consider the number of faces in the mesh. Unfortunately, there is no direct connection to the number of edges and vertices, because faces share them. However, if we split each edge into two *halfedges* (the number of which we denote by HE), we can assign three halfedges exclusively to one face (cf. Fig. 8.3a) and get:

$$\begin{aligned} & 3F = HE, && \text{since a face consists of three halfedges} \\ \Leftrightarrow \quad & \mathbf{3F = 2E}, && \text{since an edge corresponds to two halfedges} \end{aligned}$$

Inserting this in the Euler formula we obtain

$$\begin{aligned} V - E + F &= c \\ \Leftrightarrow \quad 2V - 3F + 2F &= 2c \\ \Leftrightarrow \quad 2V - F &= 2c \end{aligned}$$

Since c is negligibly small compared to V, E and F, this means

$$\mathbf{F \approx 2V}.$$

We can derive another fact if we cut the edges such that we can assign each halfedge to a vertex (cf. Fig. 8.3b). Then the ratio of halfedges and vertices tells us how many edges are adjacent to a vertex (this number is called the *valence* of that vertex). With the Euler formula we get

$$\begin{array}{rrcl} & V - E + F & = & c \\ \Leftrightarrow & 3V - 3E + 3F = 3V - E & = & 3c \\ \Rightarrow & \mathbf{E} & \approx & \mathbf{3V} \\ \Rightarrow & \mathbf{HE} & \approx & \mathbf{6V} \end{array}$$

Thus, the average valence in a triangle mesh is six; in practical applications this in fact holds. There are two typical examples of regular triangle meshes (cf. Fig. 8.3c,d): either every (interior) vertex has valence six, or the vertices alternatingly have valences four and eight.

8.3 Platonic Solids

Platonic Solids are defined to be convex polyhedra with the following properties:

- The polyhedron is closed and has genus zero.
- All faces are regular p-gons.
- All vertices have valence q.

Generally, polyhedra consisting of p-gons only and having constant vertex valence q are denoted with the *Schläfli-Symbol* $\{p, q\}$.

The interesting question is how these Platonic solids look like and how many different ones there are. We can derive this information by again splitting edges into halfedges. First, since all vertices have valence q we can assign q halfedges to each of the vertices, leading to

$$HE = 2E = qV \Leftrightarrow E = \frac{qV}{2}.$$

Conversely, we can also assign p halfedges to one face, if we split the edges like in Fig. 8.3a, and therefore

$$HE = 2E = pF \Leftrightarrow F = \frac{2E}{p} = \frac{qV}{p}.$$

Putting these equations into the Euler Formula for meshes of genus zero yields

$$\begin{array}{rrcl} & V - E + F & = & 2 \\ \Leftrightarrow & V - \frac{qV}{2} + \frac{qV}{p} & = & 2 \\ \Leftrightarrow & V & = & \frac{4p}{2p-pq+2q}, \end{array}$$

and by similar transformations we also obtain

$$E = \frac{2pq}{2p - pq + 2q} \quad \text{and} \quad F = \frac{4q}{2p - pq + 2q}.$$

At this point, consider p and q. Since Platonic solids are convex, we need them to consist of at least triangles, i.e. $p \geq 3$ (with $p = 2$ we would not be able to create a 3D object). Also, each vertex must have at least three neighbors, so $q \geq 3$. Obviously, $V, E, F > 0$ must hold as well, and since the nominators in the above equations are positive, the common denominator has to be positive as well:

$$\begin{array}{rrcl} & 2p - pq + 2q & > & 0 \\ \Leftrightarrow & (p-2) \cdot (q-2) & < & 4 \end{array}$$

With the above restrictions on p and q we obtain all possible Platonic solids (cf. Fig. 8.4):

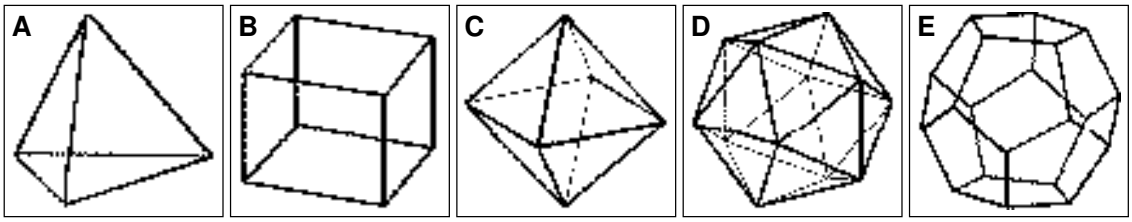


Figure 8.4: Platonic Solids: (a) $\{3,3\}$ tetrahedron, (b) $\{4,3\}$ cube, (c) $\{3,4\}$ octahedron, (d) $\{3,5\}$ icosahedron, (e) $\{5,3\}$ dodecahedron.

- *Tetrahedron* $\{3,3\}$: It is dual to itself[1].
- *Cube* $\{4,3\}$: The dual mesh is the octahedron.
- *Octahedron* $\{3,4\}$: The dual mesh is the cube.
- *Icosahedron* $\{3,5\}$: The dual mesh is the dodecahedron.
- *Dodecahedron* $\{5,3\}$: The dual mesh is the icosahedron.

8.4 Datastructures for Meshes

In order to work on polygonal meshes we have to provide suitable data structures for storing and handling them. There are two contradicting goals one might want to achieve: on the one hand one wants the structure to be as compact as possible, on the other hand one wants to traverse the mesh efficiently. In this respect it is also interesting to look at how to convert space-saving representations into traversal-efficient structures (e.g. when loading data into memory) and vice versa.

In the following discussion, we will restrict ourselves to triangle meshes, since meshes of this kind are most commonly used.

8.4.1 Compact Representations

Let us first have a look at some compact representations, i.e. datastructures that represent meshes consuming as little memory as possible. Such structures are important when saving meshes on harddisk and when transmitting them over bandwidth-limited channels (such as the internet).

The datastructures we present here are rather simple. In the lecture „Computer Graphics II“ we will discuss far superior methods when dealing with mesh compression.

Triangle Lists

A naive approach to store a mesh is to list all triangles in the mesh. For each triangle we specify its three vertices:

$$\begin{array}{rcl} M : & & \\ & v_{0,0} = & (x_{0,0},\ y_{0,0},\ z_{0,0}), \\ & v_{0,1} = & (x_{0,1},\ y_{0,1},\ z_{0,1}), \\ & v_{0,2} = & (x_{0,2},\ y_{0,2},\ z_{0,2}) \\ & \vdots & \\ & v_{m,0} = & (x_{m,0},\ y_{m,0},\ z_{m,0}), \\ & v_{m,1} = & (x_{m,1},\ y_{m,1},\ z_{m,1}), \\ & v_{m,2} = & (x_{m,2},\ y_{m,2},\ z_{m,2}) \end{array}$$

[1] A Platonic Solid with Schläfli-symbol $\{k,r\}$ is dual to the Platonic solid with Schläfli-symbol $\{r,k\}$.

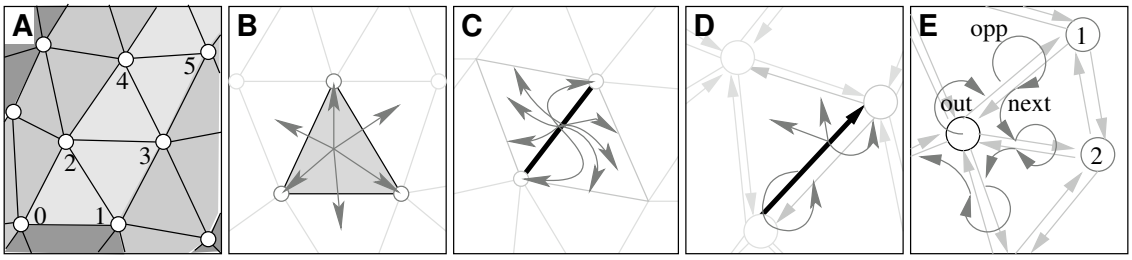


Figure 8.5: Datastructures: (a) triangle strips, (b) face based, (c) edge based, (d) halfedge based, (e) halfedge traversal.

Given a mesh with V vertices, we need obviously $2 \cdot 4 \cdot 9 \cdot V = 72V$ bytes[2] for the triangle list (recall that $F = 2V$).

Shared Vertex

The shared vertex representation avoids redundancy by first saving an array of vertex coordinates. Then, all triangles are listed, but instead of specifying the coordinates of their vertices explicitly, one refers to them by indices into the vertex array:

$$\begin{array}{rcl@{\qquad\qquad}rcl} V & : & v_0 = (x_0, y_0, z_0) & T & : & t_0 = (i_0, j_0, k_0) \\ & & \vdots & & & \vdots \\ & & v_m = (x_m, y_m, z_m) & & & t_n = (i_n, j_n, k_{n-1}) \end{array}$$

For a mesh with V vertices, we now need $3V$ floats for the vertex list, and $3 \cdot 2V$ integers for the triangle list, yielding $36V$ bytes for the mesh. This means we halved the space consumption compared to triangle lists.

Triangle Strips

In the shared vertex representation we stored the topology in a separate list. We can save this space by coding the topology implicitly into the vertex order

$$M \quad : \quad v_0 = (x_0, y_0, z_0),\ v_1,\ v_2, \ldots,\ v_m$$

by defining that each (v_i, v_{i+1}, v_{i+2}) builds a triangle $(0 \leq i \leq n)$, cf. Fig. 8.5a.

Note that in order to keep the vertex ordering (specifying triangle orientation) consistent, the actual triangles are $\Delta(v_0, v_1, v_2)$, $\Delta(v_2, v_1, v_3)$, $\Delta(v_2, v_3, v_4)$, $\Delta(v_4, v_3, v_5)$ etc. A problem with this representation is that vertices are visited twice (strips pass them at both sides), and so we still have some redundancy. Assuming we can save a mesh of F faces and V vertices using one strip only, we need $(F + 2) \cdot 3 \cdot 4 = 24 + 24V$ bytes.

Unfortunately, this assumption does not hold in practice; instead, one needs to partition the mesh into a set of strips, each strip requiring additional 24 bytes for initialization. However, the number of strips is usually small compared to the number of triangles, and therefore triangle strips turn out to be very space-efficient.

8.4.2 Representations for Efficient Traversal

For algorithms processing meshes, efficient traversal is much more important than space consumption. The operation most frequently done in algorithms is jumping from one entity to an adjacent

[2] We assume four bytes per float or integer.

entity; for example, one might traverse all vertices or faces adjacent to a given vertex (its so-called *one-ring neighborhood*). Therefore, traversal-efficient datastructures usually store references to adjacent entities.

There are three different commonly used ways to store the topology of a mesh, depending on the entity the neighbor information is built on (vertices are not suitable, since they have a varying number of neighboring entities).

Face Based

In a face based structure, each face contains pointers to its vertices and the neighboring faces (cf. Fig. 8.5b), and each vertex points to one adjacent face. In case of triangle meshes we get

$$f = [*v_0, *v_1, *v_2, *f_0, *f_1, *f_2] \quad \text{and} \quad v = [*f].$$

This representation requires $13V$ pointers for a mesh with V vertices. In order to store the whole mesh (i.e. geometry and topology) we therefore need $12V + 52V = 64V$ bytes.

There are several flaws in this representation. First, when traversing the one-ring neighborhood (rotating around one vertex), we need to check from which face we entered to find out into which face to jump next. Also, consider what happens for meshes containing general polygons of different degree. The number of neighbors of a face equals the number of edges surrounding it. This means we need to give each face as many pointers as required by the face with the maximum edge count, or we must deal with a dynamic structure. Both solutions are not ideal: either we waste memory, or we need to design a more complicated (and possibly slower) algorithm. Another drawback is that this representation lacks edges at all, i.e. this entity is not represented.

Edge Based

In the edge-based *Winged Edge* datastructure we store for each edge its two vertices, four adjacent edges (per vertex one adjacent edge in clockwise and counter-clockwise order) and the two incident faces (cf. Fig. 8.5c):

$$e = [*v_0, *v_1, *e_{0c}, *e_{0cc}, *e_{1c}, *e_{1cc}, *f_0, *f_1].$$

For both vertices and faces we store a pointer to an incident edge:

$$v = [*e] \quad \text{and} \quad f = [*e].$$

This representation requires $8 \cdot 3V + V + 2V = 27V$ pointers for a mesh with V vertices, or $12V + 108V = 120V$ bytes to store the whole mesh.

This representation requires double the space of the face based one, but it can much more efficiently store general polygon meshes. This is due to the fact that an edge has always exactly two adjacent vertices and faces. However, when traversing the mesh we still need to find out from which edge we entered in order to find the next edge.

Halfedge Based

In a halfedge based structure we split an edge p_0p_1 into two directed halfedges $\overrightarrow{p_0p_1}$ and $\overrightarrow{p_1p_0}$. For each halfedge we store the opposite halfedge, the vertex it points to (the other vertex can be obtained from the opposite halfedge), the incident face and the next halfedge (on the incident face), cf. Fig. 8.5d:

$$h = [*h_{opp}, *v, *f, *h_{next}].$$

Algorithm 7 Traversal of the vertices adjacent to v

```
h ← v.out
repeat
  output(h.vertex)
  h ← h.opp.next
until (h = v.out)
```

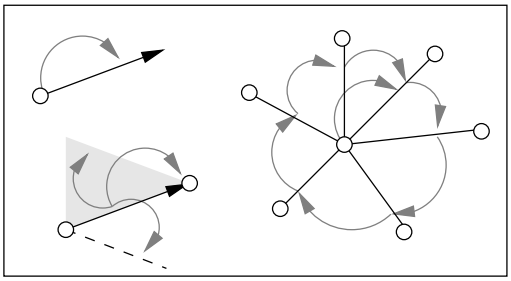
Furthermore, each vertex contains a pointer to an outgoing halfedge, and each face stores a pointer to one of the halfedges surrounding it:

$$v = [*h] \quad \text{and} \quad f = [*h].$$

This representation still requires $27V$ pointers (or $120V$ bytes) for a mesh with V vertices, but it is easy to see that traversal is very simple now. For example, in order to list all neighbors of a given vertex, we traverse the halfedges as depicted in Fig. 8.5e, leading to algorithm 7.

Constructing a Face Based Datastructure

Let us now find an algorithm which loads an unordered triangle list and builds up a face based datastructure on-the-fly. The algorithm handles the different entities in the mesh in datastructures organized as follows:



- For vertices we hold an ID, keep track of their coordinates, and store a pointer to one of the adjacent edges (above figure); $v = [x, y, z, id, *e]$. The vertices are kept in a hash-table indexed by their ID.
- For edges we store the ID of the vertex in which it ends; the vertex in which it starts references this edge either directly or by the `next`-pointers. The latter is also stored per-edge, and builds a cyclic list around the start vertex (above figure). Finally, we also store an incident face of the edge: $e = [id_v, e_{next}, f]$. Edges are referenced by the vertices.
- For faces we store pointers to their vertices and adjacent faces:

  $$f = [*v_0, *v_1, *v_2, *f_0, *f_1, *f_2]$$

  When the algorithm terminates this face based representation of the mesh will be completed.

The algorithm now proceeds as follows: it reads and stores one triangle from the list. Then, it checks whether the face's edges exist in the structure. If so, it looks up the face associated with the edge and updates the face references. This edge can now be deleted since it was referenced twice, implying that it will not be referenced anymore (two-manifold mesh). If the edge was not found, a new edge pointing to the current face is inserted into the datastructure (cf. Alg. 8).

Note that the algorithm requires $\mathcal{O}(n)$ time; a triangle has three vertices, vertex lookup in a hash-table can be considered constant, edge lookup gets pointer dereferencing, and insertion and deletion also requires constant effort.

Algorithm 8 Conversion of a Triangle List into a Face Based Datastructure

```
while not eof(trianglelist)
  read triangle Δ
  for each edge e in Δ
    find opposite triangle
      if found:  connect triangle face and delete edge
      if not found:  create new edge pointing to Δ
open edges represent mesh boundaries
```

Chapter 9

Constructive Solid Geometry

In *Constructive Solid Geometry* (*CSG*) one tries to create complex objects by combining several basic primitives using transformations and boolean operations. The object is then defined by the primitives it is built of and the sequence of operations performed on them.

The big advantage of CSG is that objects' volumes are represented by implicit functions, i.e. for any given point in space we can immediately decide whether it is inside the object or not. Additionally, we can easily combine CSG objects by boolean operations. Note that polygonal meshes approximate surfaces only. This makes it very hard to combine meshes, e.g. by union or intersection operations. Fig. 9.1 shows examples of what CSG is capable of.

The left image shows a CSG simulation of a milling process: we see the result of milling furrows into a solid using a cutter with a spherical head. The right picture shows the ACG logo: here, letters have been carved through a solid cube.

Note that CSG objects are, in contrast to polygonal meshes, descriptions of volumes. We will discuss rendering techniques for such volumetric representations in the next chapter.

9.1 Basic Primitives

In this chapter we will derive representations for the most common basic primitives used in CSG.

Since we want to combine these basic primitives using boolean operations later on, we have to determine where a given point lies with respect to the object's surface (i.e. inside, outside, or on the surface). We therefore specify an object by an *implicit function* $F : \mathbb{R}^3 \to \mathbb{R}$ (cf. Sec. 1.2),

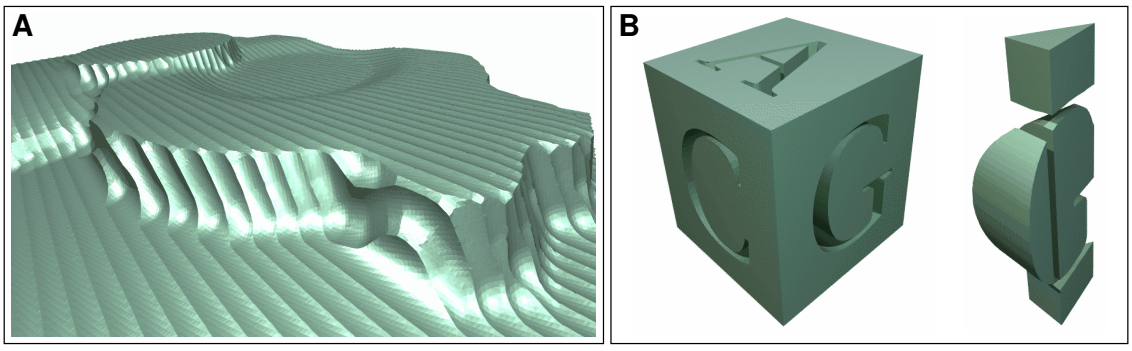


Figure 9.1: Constructive Solid Geometry: (a) milling simulation, (b) ACG logo.

such that $F(p) = 0$ if p lies on the surface of the object, $F(p) < 0$ if it is inside, and $F(p) > 0$ if it is outside. Using this function, it is easy to see that a solid primitive S is defined by

$$S = \left\{ p \in \mathbb{R}^3 \,:\, F(p) \leq 0 \right\}.$$

Besides the implicit form we will also give the *parametric* (or *explicit*) form of the surface; we can use this representation to find points on the surface.

9.1.1 Quadrics

Let us first discuss primitives which can be modeled using polynomials of degree two (these objects are also called *quadrics*), such as spheres, cylinders and cones. The implicit form of these objects has the form

$$F(x,y,z) = a_1x^2 + a_2xy + a_3xz + a_4x + a_5y^2 + a_6yz + a_7y + a_8z^2 + a_9z + a_{10}$$

for $a_1, ..., a_{10} \in \mathbb{R}$. We can write this quadratic form as a matrix-vector-product:

$$\begin{aligned} F(x,y,z) &= (x\ y\ z\ 1) \cdot \begin{pmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \\ &= ax^2 + (b+e)xy + (c+i)xz + (d+m)x + fy^2 \\ &\quad +(g+j)yz + (h+n)y + kz^2 + (l+o)z + p \end{aligned}$$

However, the above form is not unique for a given object, because some of the polynomial's coefficients a_i are the sum of two matrix coefficients a_{i_1} and a_{i_2}, and there are infinitely many choices for a_{i_1} and a_{i_2} such that $a_i = a_{i_1} + a_{i_2}$.

Taking a closer look at the matrix reveals that the upper right triangle of the matrix already suffices to specify a quadratic polynomial:

$$\begin{aligned} F(x,y,z) &= (x\ y\ z\ 1) \cdot \begin{pmatrix} a & b & c & d \\ 0 & e & f & g \\ 0 & 0 & h & i \\ 0 & 0 & 0 & j \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \\ &= ax^2 + bxy + cxz + dx + ey^2 + fyz + gy + hz^2 + iz + j \end{aligned}$$

In this form each of the matrix coefficients corresponds to exactly one of the polynomial coefficients, and therefore this form is unique for a given quadric. We can also find a unique symmetric matrix representing a given quadric:

$$\begin{aligned} F(x,y,z) &= (x\ y\ z\ 1) \cdot \begin{pmatrix} a & \frac{b}{2} & \frac{c}{2} & \frac{d}{2} \\ \frac{b}{2} & e & \frac{f}{2} & \frac{g}{2} \\ \frac{c}{2} & \frac{f}{2} & h & \frac{i}{2} \\ \frac{d}{2} & \frac{g}{2} & \frac{i}{2} & j \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \\ &= ax^2 + bxy + cxz + dx + ey^2 + fyz + gy + hz^2 + iz + j \end{aligned}$$

This form is preferable since we know from linear algebra that symmetric matrices have some properties which make it very easy to work with them.

Besides the implicit form we are also interested in the *parametric* representation of the object's surface, $f : D \to \mathbb{R}^3$. Recall that this function enumerates all points on the object's surface when traversing its range D.

We have now established a framework to specify the parametric and implicit forms of some common quadrics.

Spheres

A sphere with radius r and midpoint $(m_x, m_y, m_z)^T$ has the implicit form

$$\begin{aligned} F(x,y,z) &= (x-m_x)^2 + (y-m_y)^2 + (z-m_z)^2 - r^2 \\ &= (x\ y\ z\ 1) \cdot \begin{pmatrix} 1 & 0 & 0 & -m_x \\ 0 & 1 & 0 & -m_y \\ 0 & 0 & 1 & -m_z \\ -m_x & -m_y & -m_z & \left(m_x^2 + m_y^2 + m_z^2 - r^2\right) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \end{aligned}$$

As we already know, a sphere's surface can also be described by two angles φ and θ, therefore the parametric form of a sphere is

$$f(\varphi,\theta) = \begin{pmatrix} m_x & + & r\cos\varphi\cos\theta \\ m_y & + & r\sin\varphi\cos\theta \\ m_z & + & r\sin\theta \end{pmatrix}, \quad 0 \le \varphi \le 2\pi, \ -\frac{\pi}{2} \le \theta \le \frac{\pi}{2}.$$

Cylinders

An infinitely long cylinder through $(m_x, m_y, 0)^T$ defined by its axis $(0,0,1)^T$ and its radius r has the implicit form

$$\begin{aligned} F(x,y,z) &= (x-m_x)^2 + (y-m_y)^2 - r^2 \\ &= (x\ y\ z\ 1) \cdot \begin{pmatrix} 1 & 0 & 0 & -m_x \\ 0 & 1 & 0 & -m_y \\ 0 & 0 & 0 & 0 \\ -m_x & -m_y & 0 & \left(m_x^2 + m_y^2 - r^2\right) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \end{aligned}$$

Points on the surface can be obtained by traversing a circle of radius r at each point on the axis:

$$f(\varphi,h) = \begin{pmatrix} m_x + r\cos\varphi \\ m_y + r\sin\varphi \\ h \end{pmatrix}, \quad 0 \le \varphi \le 2\pi, \ h \in \mathbb{R}\ .$$

Cones

A cone with apex at $(m_x, m_y, m_z)^T$ opening along the z-axis with an angle of α has the implicit form

$$\begin{aligned} F(x,y,z) &= (x-m_x)^2 + (y-m_y)^2 - \tan^2\alpha (z-m_z)^2 \\ &= (x\ y\ z\ 1) \begin{pmatrix} 1 & 0 & 0 & -m_x \\ 0 & 1 & 0 & -m_y \\ 0 & 0 & -\tan^2\alpha & m_z\tan^2\alpha \\ -m_x & -m_y & m_z\tan^2\alpha & \left(m_x^2 + m_y^2 - m_z^2\tan^2\alpha\right) \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}. \end{aligned}$$

The explicit form becomes

$$f(\varphi,h) = \begin{pmatrix} m_x & + & (\tan\alpha \cdot h)\cos\varphi \\ m_y & + & (\tan\alpha \cdot h)\sin\varphi \\ m_z & + & h \end{pmatrix}, \quad 0 \le \varphi \le 2\pi, \ h \in \mathbb{R}\ .$$

9.1.2 Quartics

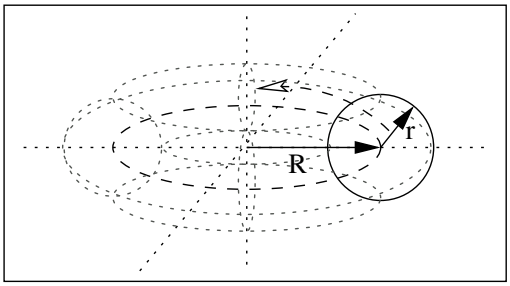
Implicit representations of degree-four polynomials (also called *quartics*) have the form

$$F(x, y, z) = a_1x^4 + a_2x^3y + a_3x^3z + a_4x^2y^2 + ... + a_{33}z + a_{34} = 0.$$

It is easy to see that there exists no matrix representation for this form; this is why we will not discuss the implicit form of quartics any further. However, we will give the explicit form of a well known quartic: the torus.

Torus

The torus is a primitive constructed by rotating a circular profile of radius r along a circular path with radius R. We will just consider a torus in the (x, y)-plane (i.e. the circular path lies in that plane) with midpoint in the origin, as depicted in the figure. The circular profile with radius r is orthogonal on the path plane, and its midpoint must be on that path (for example at $(R, 0, 0)^T$).



Such a circle can be written as

$$f_c(\theta) = \begin{pmatrix} r\cos\theta + R \\ 0 \\ r\sin\theta \end{pmatrix}, \quad 0 \le \theta \le 2\pi.$$

Now we need to recall that rotating a point $(x, y, z)^T$ by an angle of φ on the (x, y)-plane yields

$$\begin{pmatrix} x\cos\varphi - y\sin\varphi \\ x\sin\varphi + y\cos\varphi \\ z \end{pmatrix}.$$

Letting the circle defined by f_c rotate builds the torus:

$$f(\varphi, \theta) = \begin{pmatrix} (R + r\cos\theta)\cdot\cos\varphi \\ (R + r\cos\theta)\cdot\sin\varphi \\ r\cdot\sin\theta \end{pmatrix}, \quad 0 \le \varphi, \theta \le 2\pi.$$

9.2 CSG Operations

Now that we have representations for some basic primitives we can take a look at how to operate on them. There are two operations we will perform on CSG objects: transformations and boolean combinations. In the following sections we will describe how to do this, restricting ourselves to quadrics for matters of simplicity.

Obviously, an object is defined by the basic primitives it has been built of and the order and type of operations performed on them. Therefore, a CSG object can be represented by its *CSG tree*: the leaves are labelled with basic primitives, the nodes are labelled with operations, and the root is labelled with the final object. As we will see, transformations are unary operations and combinations are binary operations. Thus, a given non-leave node has at most two children, and since the binary operations are not commutative, we need them to be ordered.

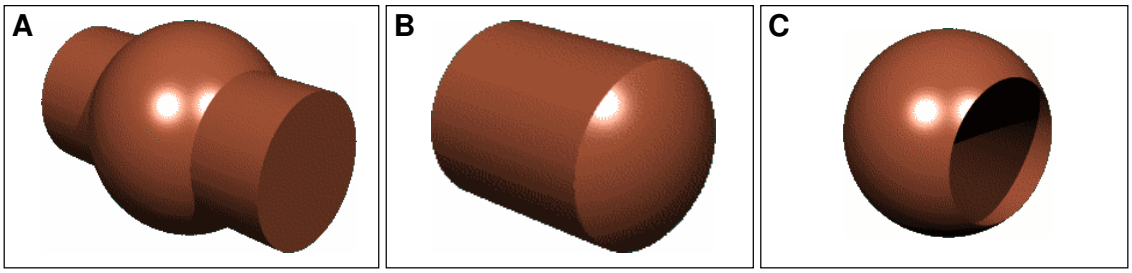


Figure 9.2: Boolean combination of a cylinder and a sphere: (a) union, (b) intersection, (c) subtraction.

9.2.1 Transformations

Transformations are unary operations on objects. Using them we can, for example, change the orientation, position and size of objects. We can use the very same transformation matrices we already discussed in Sec. 2.1.1. The remaining question is how to find an implicit representation for the transformed CSG object.

So, let $Q \in \mathbb{R}^{4\times4}$ be a quadric, and $M \in \mathbb{R}^{4\times4}$ the transformation matrix to be applied to it. Recall that the implicit function of a quadric has the form

$$v^T \cdot Q \cdot v = 0, \quad v \in \mathbb{R}^4.$$

We are interested in finding the quadric $Q' \in \mathbb{R}^{4\times4}$ such that the transformed object will have the implicit form

$$v^T \cdot Q' \cdot v = 0, \quad v \in \mathbb{R}^4.$$

Note that transforming the object by the matrix M is the same as applying the inverse transformation M^{-1} on the points we feed into the function (e.g. for scaling we get the same relative result no matter whether we scale the object or inversely scale the space, cf. Sec. 2.1.3). Therefore

$$\begin{aligned} 0 &= v^T \cdot Q' \cdot v \\ &= \left(M^{-1}v\right)^T \cdot Q \cdot \left(M^{-1}v\right) \\ &= v^T \cdot \left[\left(M^{-1}\right)^T \cdot Q \cdot M^{-1}\right] \cdot v. \end{aligned}$$

So, the quadric of the transformed object is

$$Q' = \left(M^{-1}\right)^T \cdot Q \cdot M^{-1}.$$

9.2.2 Combinations

When combining two solids given by their implicit functions F_1 and F_2 we operate on two sets of points

$$S_1 = \left\{p \in \mathbb{R}^3 \,:\, F_1(p) \le 0\right\} \quad \text{and} \quad S_2 = \left\{p \in \mathbb{R}^3 \,:\, F_2(p) \le 0\right\}.$$

In order to create a new object S (represented by an implicit function F) we can perform boolean operations on the points:

- *Union* (cf. Fig. 9.2a): S is composed of the points in S_1 and those in S_2.

$$\begin{aligned} S = S_1 \cup S_2 &= \left\{p \in \mathbb{R}^3 \,:\, F_1(p) \le 0 \vee F_2(p) \le 0\right\} \\ &= \left\{p \in \mathbb{R}^3 \,:\, \min\left\{F_1(p),\, F_2(p)\right\} \le 0\right\} \\ \Rightarrow \qquad F(p) &= \min\left\{F_1(p),\, F_2(p)\right\} \end{aligned}$$

- *Intersection* (cf. Fig. 9.2b): S is composed of the points which are both in S_1 and S_2.

$$\begin{array}{rrcl} & S = S_1 \cap S_2 & = & \left\{p \in \mathbb{R}^3 \,:\, \max\left\{F_1(p),\, F_2(p)\right\} \leq 0\right\} \\ \Rightarrow & F(p) & = & \max\left\{F_1(p),\, F_2(p)\right\} \end{array}$$

- *Subtraction* (cf. Fig. 9.2c): S is composed of the points which are in S_1 but not in S_2:

$$\begin{array}{rrcl} & S = S_1 \setminus S_2 & = & \left\{p \in \mathbb{R}^3 \,:\, \max\left\{F_1(p),\, -F_2(p)\right\} \leq 0\right\} \\ \Rightarrow & F(p) & = & \max\left\{F_1(p),\, -F_2(p)\right\} \end{array}$$

Chapter 10

Volume Rendering

After having discussed how to build volumetric objects in the last chapter, we will now present methods for visualizing them. *Volumetric objects* are represented by implicit functions or may also be discretized to a regular 3D grid of scalar values (cf. Sec. 10.1).

Direct methods render a volumetric object based on its implicit function, i.e. for a given viewing position they directly generate a 2D image of the volume.

Indirect methods instead first extract an iso-surface from the volumetric data (represented by a polygonal mesh) and render this mesh using standard techniques.

After discussing volumetric representations in more detail we will discuss both rendering approaches.

10.1 Volumetric Representations

The volume of an object is represented by an implicit function

$$F : \mathbb{R}^3 \to \mathbb{R},$$

mapping points in 3-space to an object-dependent value. Common approaches to interpret this value include the following:

Implicit Functions: The surface S of an object is defined to be the kernel of the function F:

$$S = \left\{ x \in \mathbb{R}^3 \,:\, F(x) = 0 \right\}.$$

Usually, one also defines the interior of the object to be $\left\{ x \in \mathbb{R}^3 \,:\, F(x) < 0 \right\}$, and its exterior to be $\left\{ x \in \mathbb{R}^3 \,:\, F(x) > 0 \right\}$. In the CSG approach described in the previous chapter we used implicit functions of that type.

Signed Distance Functions are a special case of general implicit functions. Their function values not only represent inside/outside information (the sign), but also the distance to the object's surface S (the absolute value):

$$F(x) = \min_{p \in S} \|p - x\| \cdot \begin{cases} +1 & \text{if } x \text{ is outside the volume defined by } S \\ -1 & \text{otherwise} \end{cases}$$

Value Functions return some scalar property value for a given 3D point, for example its density value or temperature. From this type of function one can extract a surface for a given function value v:

$$S_v = \left\{ x \in \mathbb{R}^3 \,:\, F(x) = v \right\}$$

Such surfaces S_v are called *iso-surfaces*, since they consist of all points in space having the same specific function value v. For instance, medical imaging (CT, MRT) generates such functions; given a point in space, they specify the density of the tissue at that position. Choosing a density d of a certain organ allows the corresponding iso-surface S_d to be extracted or directly rendered. Note that $F_v(x) := F(x) - v$ is the implicit function representing the iso-surface S_v.

Since in most cases these functions are obtained by some measuring process, we will generally not have the continuous function F, but instead work on a discrete (regular) grid of sample values $F_{i,j,k}$:

$$F_{i,j,k} := F(i\Delta x, j\Delta y, k\Delta z).$$

However, if we need a continuous function we can reconstruct an approximation from the set of sample values, e.g. by *trilinear interpolation*: Given some point $(x, y, z)^T \in \mathbb{R}^3$ we interpolate the function values of the neighboring grid points in order to estimate $F(x, y, z)$. Assuming w.l.o.g. that we have a uniform grid with sample distance 1 (i.e. $\Delta x = \Delta y = \Delta z = 1$) and $(x, y, z)^T \in [0, 1]^3$, we get

$$\begin{aligned} F(x,y,z) \;=\; & F_{0,0,0} \cdot (1-x)(1-y)(1-z) \\ & + F_{1,0,0} \cdot x(1-y)(1-z) \\ & + F_{0,1,0} \cdot (1-x)y(1-z) \\ & + F_{1,1,0} \cdot xy(1-z) \\ & + F_{0,0,1} \cdot (1-x)(1-y)z \\ & + F_{1,0,1} \cdot x(1-y)z \\ & + F_{0,1,1} \cdot (1-x)yz \\ & + F_{1,1,1} \cdot xyz \end{aligned}$$

In this context, the grid points $F_{i,j,k}$ (or sometimes the cubes inbetween them) are also referred to as *voxels* (volume elements) — similar to pixels (picture elements).

10.2 Direct Volume Rendering

One way to visualize the volume is to render it directly based on the implicit function defining it. In contrast to *indirect methods*, discussed in the next section, these algorithms are capable of visualizing the whole volume (usually in a semi-transparent manner), instead of just one iso-surface. There are two different approaches for direct rendering:

- *Backward-Mapping* (from image space to object space): For each pixel in the image the algorithm computes its color by considering the viewing ray through that pixel (corresponding to the back-projection of that pixel into 3-space).
- *Forward-Mapping* (from object space to image space): These algorithms forward-project each voxel (grid point) onto the image plane in order to composite their contributions into the final image.

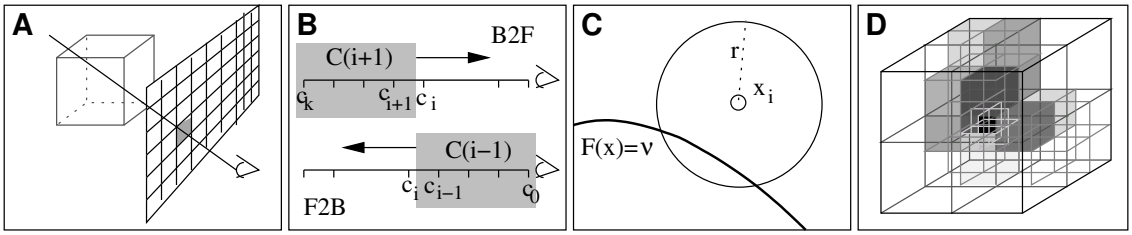


Figure 10.1: Ray Casting: (a) viewing rays, (b) back-to-front vs. front-to-back compositing, (c) discrete surface visualization, (d) octree.

The methods we will discuss in the next sections can be classified w.r.t. these two types as follows:

- Backward mapping:
  - Volume Ray Casting
- Forward mapping:
  - Volume Splatting
  - Shear Warp Factorization
  - Volume Slicing

10.2.1 Volume Ray Casting

Volume Ray Casting [Levoy, 1988, Levoy, 1990] is a method to visualize volumetric objects by sending rays from the eye into the scene and collecting color contributions of voxels traversed by these rays (cf. Fig. 10.1a). Hence, ray casting is the standard example of a backward-mapping algorithm. Since we perform an action for each image pixel, this is also an image-order (as opposed to an object-order) algorithm.

Volume Rendering Integral

Let us first consider the case in which we want to visualize the whole volume of the object in a semi-transparent manner; other rendering modes will be discussed later. We will use two functions for ray casting:

$$\begin{aligned} c : \mathbb{R}^3 \rightarrow RGB \quad & \text{The colors of the points} \\ \mu : \mathbb{R}^3 \rightarrow \mathbb{R}^+ \quad & \text{The opacities of the points} \end{aligned}$$

Given an implicit function F (or a grid of scalar values) we could e.g. set $\mu(x) = F(x)$ and choose some color $c(x)$ depending on the value (density) of $F(x)$ (in the simplest case, we would make c a constant function).

We compute the colors of screen pixels by shooting viewing rays through them and summing up the color contributions of all the voxels passed along the ray. So, let $L(s)$, $s \geq 0$, be this viewing ray such that $L(0)$ is the viewer's position.

In order to compute the contribution of some point $L(s)$ to the pixel color we have to take into account that its color $c(L(s))$ gets damped by all the points in front of it. This attenuation factor apparently depends on the total opacity of these points and can be modeled by:

$$e^{-\int_0^s \mu(L(x))\,dx}.$$

Integrating the contributions of all points along the ray yields the final pixel color C:

$$C = \int_0^\infty c(L(s)) \cdot e^{-\int_0^s \mu(L(x))\,dx}\,ds.$$

This equation is called the *Volume Rendering Integral*. When implementing ray casting it is pretty clear that this integral cannot be evaluated exactly. In order to compute the screen pixel's color we therefore discretize this expression: we limit the length of the viewing ray and only pick k samples on the resulting ray segment:

$$\{s_i = (c_i, \mu_i) \,:\, 0 \leq i \leq k\}$$

(we shall define s_0 to be nearest to the viewer). Discretizing both integrals to finite sums over these samples yields

$$C = \sum_{i=0}^{k} \left(c_i \cdot e^{-\sum_{j=0}^{i-1} \mu_j} \right) = \sum_{i=0}^{k} \left(c_i \cdot \prod_{j=0}^{i-1} e^{-\mu_j} \right).$$

If we define opacity values α_i to be $(1 - \alpha_i) = e^{-\mu_i}$, we get the final discretized version of the volume rendering integral:

$$C = \sum_{i=0}^{k} \left(c_i \cdot \prod_{j=0}^{i-1} (1 - \alpha_j) \right).$$

Discrete Compositing

The last equation can efficiently be evaluated iteratively in either back-to-front or front-to-back order. As a kind of normalization the final sample s_k is usually set to a fully opaque background color, i.e. $c_k = c_{bg}$ and $\alpha_k = 1$; this takes into account that the volume is finite and surrounded by a scene.

Note that in the following blending equations we will use *opacity-weighted* colors $\hat{c}_i := \alpha_i c_i$, i.e. colors that are pre-multiplied by their opacities. This provides a convenient notation and also can be shown to avoid interpolation artifacts. Therefore the expression to be evaluated gets

$$\hat{C} = \sum_{i=0}^{k} \left(\hat{c}_i \cdot \prod_{j=0}^{i-1} (1 - \alpha_j) \right).$$

In this formula the product $\prod_{j=0}^{i-1}(1-\alpha_j)$ can be computed iteratively by summing up the opacities while traversing the samples. We can distinguish two different ways of doing this compositing (cf. Fig. 10.1b):

- *Back-to-front:* We can accumulate the colors stepping from s_k to s_0. In each step the old color gets damped by the opacity of the current sample (the final opacity-weighted color is $\hat{C} = \hat{C}_0$). Starting with $\hat{C}_k = \hat{c}_k = \alpha_k \cdot c_k$ we get

$$\hat{C}_i \;=\; \hat{c}_i + (1 - \alpha_i)\hat{C}_{i+1}, \quad i = \{k-1, \ldots, 0\}.$$

- *Front-to-back:* We can also accumulate colors starting from sample s_0. We then also have to accumulate opacity values in A_i. Starting with $\hat{C}_0 = \hat{c}_0$ and $A_0 = \alpha_0$ the iteration is

$$\begin{array}{rcl} \hat{C}_i & = & \hat{C}_{i-1} + (1 - A_{i-1})\hat{c}_i \\ A_i & = & A_{i-1} + (1 - A_{i-1})\alpha_i \end{array}, \quad i = \{1, \ldots, k\},$$

  where the A_i accumulate opacities: $(1 - A_i) = \prod_{j=0}^{i-1}(1 - \alpha_j)$.

The resulting ray casting algorithm using front-to-back compositing is shown in Alg. 9.

Algorithm 9 Ray Casting.

```
for all pixels p
  find ray from eye e through p:   L : e + λd
  init Ĉ(p) = A(p) = 0
  for all samples s = e + λ_i d, i = {0, ..., k}
    get c(s) and α(s) by trilinear interpolation
    Ĉ(p) += (1 − A(p)) · α(s) · c(s)
    A(p) += (1 − A(p)) · α(s)
```

Iso-Surface Visualization

Up to now we only considered the semi-transparent visualization of a whole volume dataset. However, volume ray casting can also be used to visualize iso-surfaces S_v corresponding to a scalar value v. We can achieve this by assigning full opacity to points x on the iso-surface (having $F(x) = v$), and full transparency otherwise:

$$\mu(x) = \begin{cases} 1 & \text{if } F(x) = v \\ 0 & \text{otherwise} \end{cases}$$

Using an ideal ray casting method, all points on the iso-surface would be rendered and potentially lighted. However, we evaluate the function F at several sample points along the viewing rays only. If a sample point does not happen to exactly lie on the iso-surface, we would not detect the ray intersection at all. Therefore, pixels might not be colored even though the corresponding viewing ray crosses the iso-surface.

In order to work around the problem of sampling density, we weaken the above condition. We define a narrow band around the iso-surface and assign non-transparent α values to all points within this band. If the thickness of this band is r we want to assign (a high) opacity α_v for points on the iso-surface ($F(x) = v$). For points near the iso-surface the opacity should blend to 0, depending on the deviation $|F(x) - v|/r$ (cf. Fig. 10.1c).

Since the narrow band should have constant thickness, we also have to take the gradient of F into account. Recall from analysis that the gradient (more precisely its norm) measures how fast function values change at this point. This leads to the following opacity function for sample points x:

$$\alpha(x) = \alpha_v \cdot \begin{cases} 1 & \text{if } \|\nabla F(x)\| = 0 \ \wedge \ F(x) = v \\ 1 - \frac{1}{r}\frac{|F(x)-v|}{\|\nabla F(x)\|} & \text{if } \|\nabla F(x)\| > 0 \ \wedge \ |F(x) - v| < r\,\|\nabla F(x)\| \\ 0 & \text{otherwise} \end{cases}$$

Volume Shading

In order to achieve better visual results and enhance the three-dimensional impression we can include lighting into the volume rendering method. This is especially important for visualizing iso-surfaces. As we have seen in Sec. 2.3, we have to provide normal vectors associated with the points in order to use the Phong lighting model.

The normal of any point in the volume is just the gradient of the implicit function at that point:

$$N(x, y, z) = \frac{\nabla F(x, y, z)}{\|\nabla F(x, y, z)\|},$$

where the gradient is the vector containing the partial derivatives

$$\nabla F = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z}\right)^T.$$

If we are not given a continuous function F, but a grid of scalar values $F_{i,j,k}$ instead, we can approximate the gradient using central differences:

$$(\nabla F)_{i,j,k} = \frac{1}{2} \cdot \begin{pmatrix} F_{i+1,j,k} - F_{i-1,j,k} \\ F_{i,j+1,k} - F_{i,j-1,k} \\ F_{i,j,k+1} - F_{i,j,k-1} \end{pmatrix}.$$

Optimizations

It is easy to see that ray casting is computationally quite costly. Therefore, one tries to eliminate at least some calculations by optimizing parts of the algorithm.

Spatial Data Structures One can use spatial data structures in order to accelerate the ray traversal; an example of such a structure is the *octree* (cf. Fig. 10.1d). Each node in an octree represents a cube, and has as its children its eight octants. In case of ray casting we would recursively partition the voxel grid into octants, and store at each node the range of function values F we encounter in the octant. Using this data-structure, we can optimize the calculations for sample points on the viewing rays:

- We need not consider empty regions in the grid, since the ray passes them unaltered. Therefore, sampling points falling into empty octree cells can be ignored.
- In case we are rendering the iso-surface corresponding to a function value v, we can obviously also skip all cells the range of which does not contain v.
- In octree cells with almost constant function values (homogeneous regions) we can use a lower sampling rate for the ray.

Early Ray Termination We can also terminate sampling along the ray depending on the current opacity value: It is clear that if the opacity value reaches a certain threshold at a sample point, the contribution of samples behind that point is so small that we can ignore it. Therefore, when using front-to-back compositing, we can stop traversing the ray any further as soon as the accumulated opacity value reaches a certain threshold [Levoy, 1990].

Fast Cell Traversal Further improvement can be achieved by accelerating the calculation of the next sample point or the next voxel cell along the ray: We can iteratively calculate these points/cells by extending the Bresenham algorithm (cf. Sec. 3.3.2) to the three-dimensional case. Instead of finding the next pixel in the line case, the 3D variant will find the next voxel to be traversed.

10.2.2 Volume Splatting

The idea behind *Splatting* [Westover, 1990] is that each point in the voxel grid provides a certain contribution to the final color appearing on the screen. Thus, we can associate each grid point with a spherical color splotch, and project this splotch onto the screen.

These color splotches are constructed by taking the color value $c(v)$ of the corresponding grid point v and blurring or extending it using a spherical filter kernel, i.e. we damp the color $c(v)$ with growing distance to the grid point v. Then, the color that v contributes to a 3D point $(x, y, z)^T$ gets

$$contrib_v(x, y, z) = h(x - v_x, y - v_y, z - v_z) \cdot c(v).$$

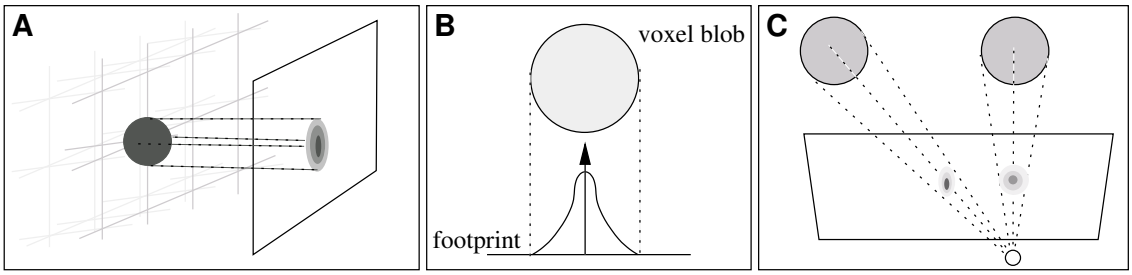


Figure 10.2: Splatting: (a) splatting a splotch, (b) footprint, (c) perspective projection.

The attenuation function h needs to be non-negative, and it should take its maximum one for the actual grid point, i.e. $h(0,0,0) = 1$. For example, we could use a quadratic attenuation function to get a filter kernel of radius r by

$$h(x,y,z) = \begin{cases} 1 - \frac{x^2+y^2+z^2}{r^2} & \text{if } x^2+y^2+z^2 \leq r^2 \\ 0 & \text{otherwise} \end{cases}$$

Let us consider an orthogonal projection of this splotch onto the screen (cf. Fig. 10.2a). We can assume w.l.o.g. that the projection direction equals the z-axis. Then

$$contrib_v(x,y) = c(v) \cdot \int_{-\infty}^{\infty} h(x - v_x, y - v_y, z)\, dz.$$

Note that the shape of the intensity distribution is the same for all grid points, and it is called the *footprint* of the grid point (cf. Fig. 10.2b):

$$footprint(x,y) = \int_{-\infty}^{\infty} h(x,y,z)\, dz.$$

In general, the projection will not be orthogonal but perspective. Then, the shape of the footprint depends on the angle between the viewing axis and the projection ray of the grid point (cf. Fig. 10.2c). Additionally, it also depends on the distance between viewer and grid point, but this dependence can be expressed by a linear scaling factor. Since we do not really need the exact footprint, we can pre-calculate approximate footprints for a set of discrete angles and store them in a table. When we need to find the footprint for a given grid point, we just pick the closest hit in that table. Having the footprints of all grid points, we just traverse the grid points and accumulate their footprints at the screen pixels from back to front.

Compared to ray casting the accumulation of the footprints is a much easier operation than the traversal of the viewing ray samples, since the footprints can be looked up in the footprint-table. The required filtering simplifies to a 2D convolution, as opposed to 3D filtering for the volume sampling along a ray. However, we have to establish a back-to-front ordering for this splatting in order to get correct accumulation results. Since this ordering depends on the current viewing transformation, it has to be re-done for each frame.

Referring to the taxonomy of direct methods we stated on page 102 earlier in this chapter, splatting is a *forward-mapping* algorithm, since voxels are projected onto the image plane. It is also classified to be an object-order algorithm, since we perform some action for each voxel (instead of performing an action for each pixel in image-order algorithms).

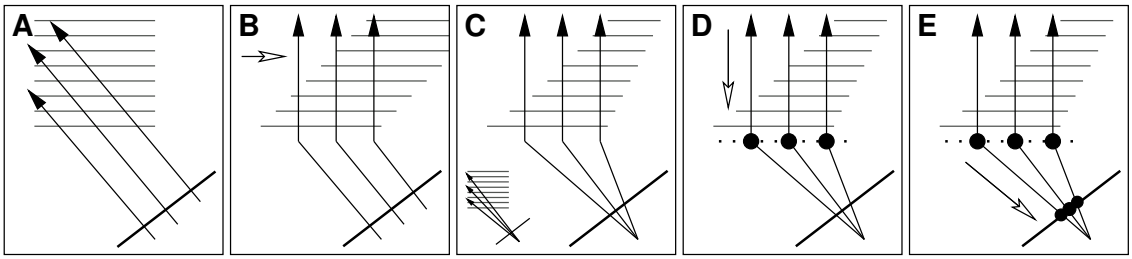


Figure 10.3: Shear Warp Factorization: (a) orthogonal projection, (b) shearing, (c) perspective projection, (d) compositing, (e) warping.

10.2.3 Shear Warp Factorization

Consider the voxel grid to be an ordered set of 2D-slices. It easy to see that arranging these slices parallel to the image plane greatly simplifies rendering and compositing. Assuming an orthogonal projection along the z-axis the pixels (x, y) of all slices project to the same image plane pixel. Therefore finding color samples along a ray simplifies to enumerating the color values $F_{x,y,i}$ for all slices $i = \{0, \ldots, N\}$.

The *Shear Warp Factorization* [Lacroute and Levoy, 1994] makes use of this fact by reducing any given viewing position to that basic case. This can be achieved by shearing the grid along its principal axes such that the viewing rays become parallel to one of the grid directions (cf. Fig. 10.3a,b). For a perspective projection we additionally have to scale the slices in order to take perspective foreshortening into account (cf. Fig. 10.3c).

Since shearing and scaling is done along the principal grid axes, each slice can be transformed on its own. This means we just have to perform 2D image transformations for each slice, which is much more efficient than doing three-dimensional image re-sampling.

The main axis along which to slice and shear should be the axis most parallel to the viewing direction. In order to further simplify the algorithm we first apply a permutation of the coordinate axes that maps each viewing direction to the case where the slicing direction is z.

After this permutation the shearing and scaling of slices is performed. Then we can composite all slices onto an intermediate image plane in sheared space that is parallel to the slices (cf. Fig. 10.3d). The required compositing is now trivial: first, the viewing rays are orthogonal to the slices, and second, the image scanlines are parallel to the volume scanlines. Exploiting this scanline coherence, the compositing can be implemented very efficiently.

The last step is mapping the composited intermediate image to the final image plane. Since perspective foreshortening has already been taken care of, this is just an affine warping (cf. Fig. 10.3e). This operation again works on 2D images and therefore is very efficient.

Putting it all together, the final image can be obtained by factorizing viewing transformation and projection into

$$M_P M_V = W \cdot C \cdot S \cdot P,$$

where the matrices represent the steps of the Shear Warp Factorization:

- M_P and M_V are the projection and the viewing matrix, respectively.
- P permutes the grid axes such that shearing can be performed on the (x, y)-slices.
- S shears and scales the slices such that the viewing rays get perpendicular to them.
- C represents the orthogonal projection and compositing onto the intermediate image-plane.

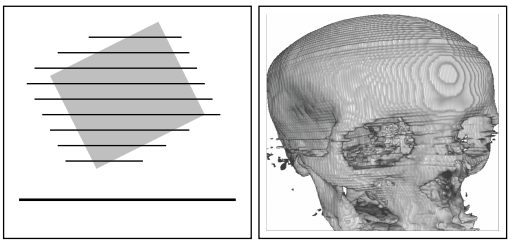


Figure 10.4: Volume Slicing: Several 3D-textured quad result in slices through the volume (left). Blending them back-to-front yields the final image (right).

- W performs the final warping of the intermediate image to the viewport. Since projection and viewing matrix are known, the matrix W can be derived by $W = M_P M_V \left(CSP\right)^{-1}$.

One of the main advantages of Shear Warp Factorization is that it exploits scan-line coherence and simplifies 3D image transformations to more efficient 2D operations. In contrast to ray casting, where we traversed the grid along a viewing ray, the voxels are not addressed randomly in this approach. This consequently enables the exploitation of spatial locality in the underlying array structure.

10.2.4 Volume Slicing

Volume Slicing [Westermann and Ertl, 1998] is a volume rendering approach that maps most of its sub-tasks to the graphics hardware. Although slicing is a rather complex algorithm, more brute-force than elegant, the hardware acceleration of modern GPUs makes it the most efficient method for interactive volume rendering nowadays.

The basic idea is to represent the volume dataset by a 3D texture (cf. Sec. 4.5.3). Drawing a textured quad (providing correct texture coordinates) results in a slice through the volume. We slice the complete volume by constructing many of these quads that are oriented parallel to the image plane. Blending these slices yields the final image (cf. Fig. 10.4).

Most of the steps required for this approach can be done by the graphics hardware:

- Viewing transformation and projection are done by the graphics hardware.
- Sampling the volume using trilinear interpolation is done by the texture unit.
- Compositing is done by alpha-blending.
- Volume shading using the Phong lighting model can be done by pixels shaders.
- Transfer functions assigning color and opacity to density values can also be implemented using pixel shaders.

All of these steps are computationally quite expensive; however, performing them using hardware acceleration makes this approach faster than other techniques, in which all computations must be performed in software.

10.3 Indirect Volume Rendering

Indirect volume rendering methods do not display the volume directly, but instead they extract an iso-surface from the volume, and display a triangle mesh representing that surface using the rendering pipeline.

As a consequence, the iso-surface is all that is rendered. We cannot see features around it, nor can we visualize the whole volume in a semi-transparent manner, as we could do using direct methods.

We will present the standard algorithm for iso-surface extraction, the so-called *Marching Cubes* (or, in the 2D case, Marching Squares). We will also discuss the Extended Marching Cubes algorithm, which does a better job in reconstructing sharp features.

10.3.1 Marching Cubes

For reasons of simplicity we will discuss the 2D counterpart, *Marching Squares*, and then shortly mention how to extend it to three dimensions, since both algorithms are almost identical.

Marching Squares solves the problem of extracting an iso-line from a 2D grid of scalar values. The volume cells (the "squares") are examined one by one and line segments are created if the iso-line passes through them. Whether the iso-line intersects a cell can be determined based on the signs the implicit function takes in the four corners of this cell (remember that the sign carries inside/outside information).

Whenever the endpoints of a cell's edge have different signs (w.r.t. the implicit function) the iso-line crosses that edge. Hence, the iso-line intersects a cell if not all of its corners have the same sign. Since each corner can take either positive or negative function values, we get $2^4 = 16$ different sign distributions for a cell. The respective line segments to be generated can therefore be stored in a table holding 16 entries. Fig. 10.5e shows the base cases; all other cases can be obtained by rotations.

Unfortunately, in some cases the signs of the corners do not uniquely identify the orientation of the line segments (see case 4 for an example). One can disambiguate these cases by looking at the function value of the square's midpoint. If we only know the function values at the grid points, other function values can be obtained by bilinear interpolation.

Having determined the orientation of the line segments (topology), we still need to estimate the intersection positions of the iso-line with the cell edges, yielding the endpoints of the line segments (geometry). These intersection points are estimated by linear interpolation of the edge's endpoints based on the function values of the implicit function. Consider an edge $\overline{p_0 p_1}$ crossing the iso-line, i.e. $F(p_0) < 0$ and $F(p_1) > 0$. The point of intersection p is the zero-crossing of F (cf. Fig. 10.5b), that is approximated by

$$p = p_0 + \frac{|F(p_0)|}{|F(p_1)| + |F(p_0)|}(p_1 - p_0).$$

Note that since neighboring squares share a common edge, the resulting segments match together and build one or more polygons (cf. Fig. 10.5c).

Marching Cubes [Lorensen and Cline, 1987] works very similar. It traverses all cubes in the grid (voxel cells) and checks whether the corresponding eight vertices are equal in sign. If not, the iso-surface intersects this cell and the algorithm generates a patch of up to four triangles. The triangle configuration is looked up in a table, which this time contains $2^8 = 256$ entries; Fig. 10.5f shows the 15 base cases. Note that again some of these cases are ambiguous (e.g. case 5); we can again resolve this by checking the sign inside the cube (trilinear interpolation). The endpoints of the triangles, i.e. the edge intersections, are again calculated using linear interpolation along the edges.

One of the advantages of Marching Cubes is that it is local: cubes are processed one-by-one based on local information only (function values at cube corners). Therefore, it is easily possible to parallelize Marching Cubes. This together with the fact that the operations performed are quite simple makes Marching Cubes a very fast algorithm for iso-surface extraction.

However, consider what happens with large flat iso-surfaces: the algorithm triangulates cell by cell, leading to a very fine triangulation and to an unnecessarily complex resulting mesh. Also,

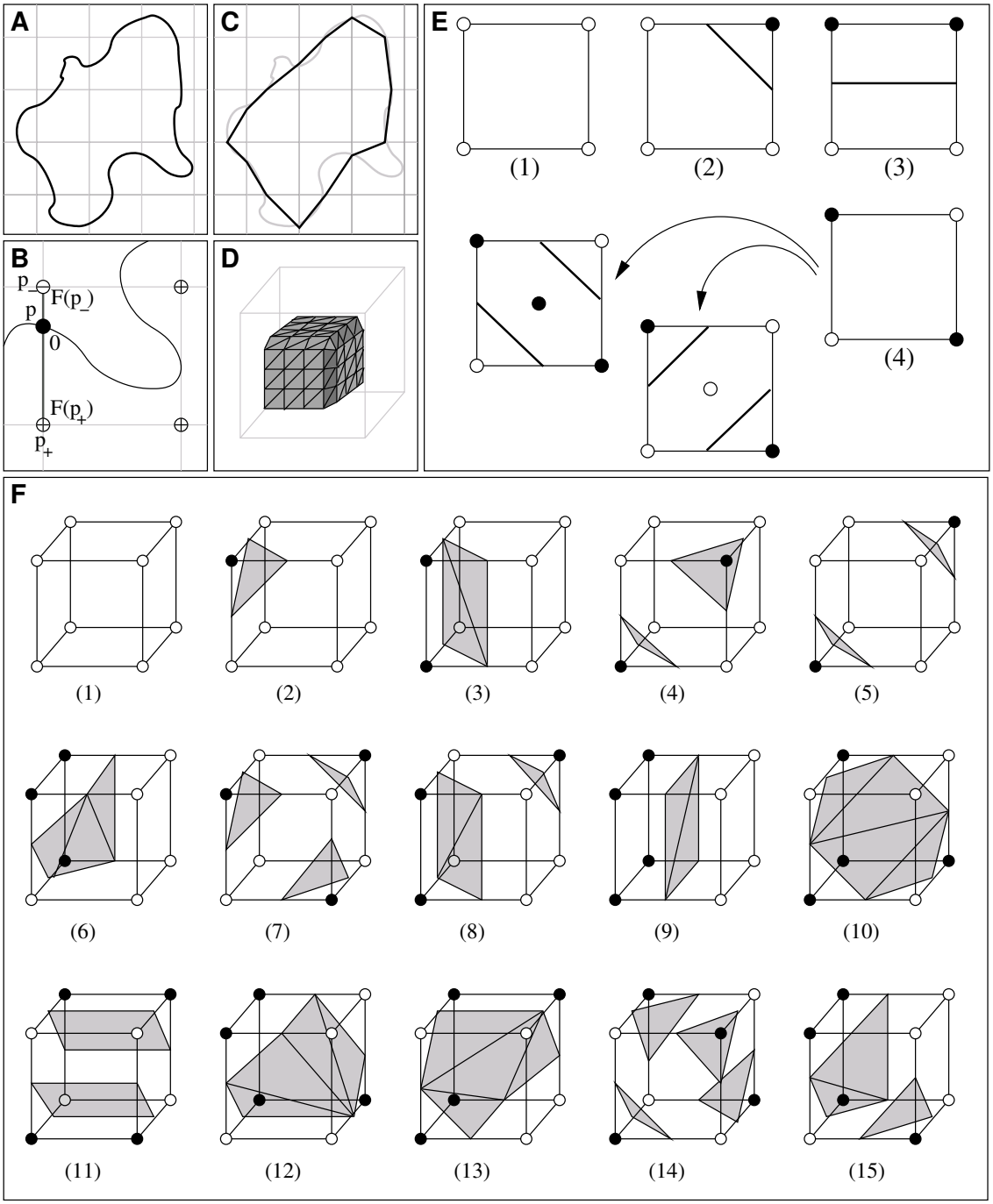


Figure 10.5: Marching Squares and Marching Cubes: (a) object, (b) intersection point calculation, (c) Marching Squares result, (d) Marching Squares configurations, (e) Marching Cubes configurations.

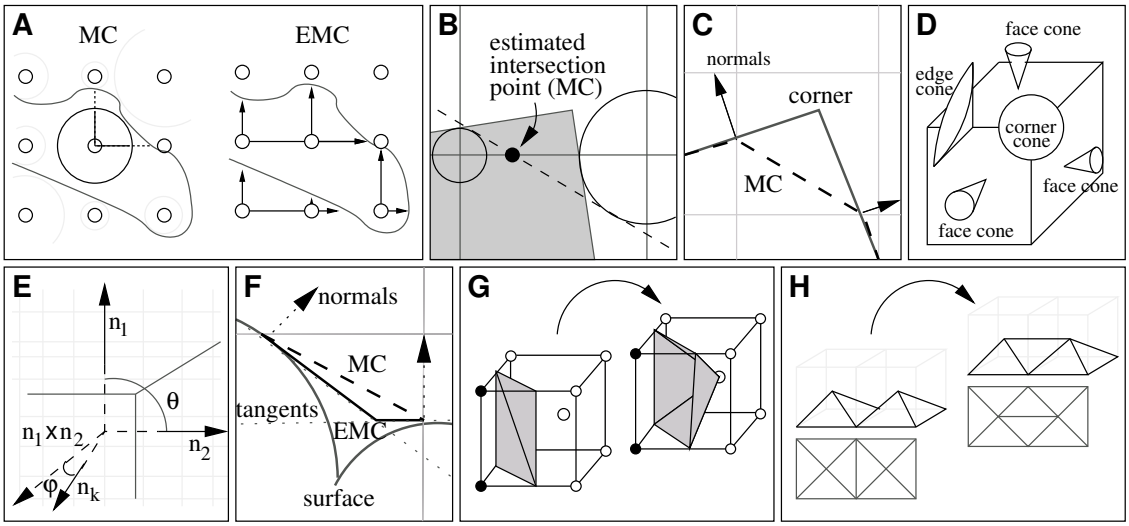


Figure 10.6: Extended Marching Cubes: (a) distance values, (b) badly approximated intersection using Marching Cubes, (c) feature cutting, (d)(e) feature detection, (f) improvement, (g) feature sampling, (h) feature reconstruction.

the resulting triangles might be badly shaped (thin and long). We can solve these problems by applying post-processing steps to the resulting mesh; such operations include *mesh decimation* and *mesh optimization* (as discussed in ,,Computer Graphics II“).

10.3.2 Extended Marching Cubes

Compared to Marching Cubes (MC), the Extended Marching Cubes (EMC) [Kobbelt et al., 2001] improves the calculation of the intersection points and the surface reconstruction. For simplicity, the figures mostly refer to the Extended Marching Squares algorithm, the 2D-equivalent of EMC.

One severe drawback of MC is that it provides a very bad approximation at sharp features like edges or corners (cf. Fig. 10.8d). This is mainly because of two reasons: the grid's distance values provide no directional information and sharp features within the cells are not detected. The Extended Marching Cubes solves both problems and therefore provides a much higher quality for the reconstruction of sharp features.

Directed Distances

Recall that in MC we have one (signed) distance value for each grid point. This value corresponds to the minimum distance between the grid point and the surface, regardless of the direction in which this distance is encountered. When computing the intersection points, we use this value to approximate the zero-crossing, although the distance between grid point and surface along the cube's edges is most probably not the minimum distance (cf. Fig. 10.6a). The more the distances differ the worse the resulting approximation gets (cf. Fig. 10.6b).

To alleviate this, the EMC stores three directed distance values for each grid point, corresponding to the exact distance between grid point and surface along the cube's edges. By doing so, at the cost of triple space consumption, we can compute the exact intersection points rather than just approximate them.

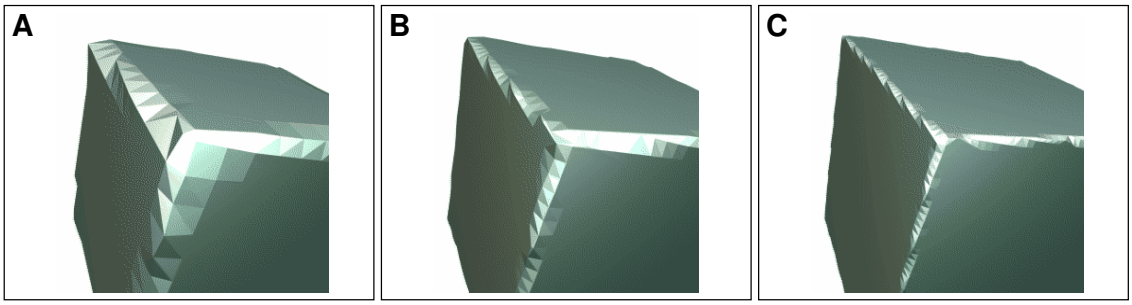


Figure 10.7: Alias errors in surfaces generated by MC due to the fixed sampling grid.

Feature Reconstruction

Consider what happens when sharp features (like edges or corners) of the original surface lie inside a voxel cube. With MC, we compute the intersection points on the cube's edges and simply connect them according to the pattern we look up in the table. In the resulting mesh sharp features of the original object are chopped off (cf. Fig. 10.6c).

Since MC computes samples as edge intersections only, all sample points lie on a regular grid, therefore resulting in alias problems. By increasing the grid resolution, this effect becomes less and less visible, but the problem is not really solved since the normal vectors of the reconstructed surface do not converge to the normals of the iso-surface (cf. Fig. 10.7).

The only way to get rid of these alias artifacts is to place sample points on the sharp features. EMC detects sharp features (detection), computes new samples on these features (sampling), and integrates them into the mesh (reconstruction).

Detection Features within a cube can be detected by considering the surface normals n_i at the intersections p_i of the cube's edge and the iso-surface. These normals are the gradients of the implicit function at the intersection points:

$$n_i = \frac{\nabla F(p_i)}{\|\nabla F(p_i)\|}$$

Based on these normals, EMC detects features inside the cubes. Consider the cone spanned by all the normals n_i (cf. Fig. 10.6d). If all intersection points p_i lie on a flat region (i.e. there is no sharp feature), the normals will be almost parallel, and the cone spanned by them will have a small opening angle. If there is a sharp feature, we will have normals from different sides of the object which vary stronger and therefore result in a larger opening angle of the cone. Hence, we consider the maximum angle between the normals n_i:

$$\vartheta := \max_{i,j} \angle(n_i, n_j)$$

If $\vartheta \geq \vartheta_{sharp}$ (ϑ_{sharp} being a threshold angle), we can conclude that we detected a sharp feature. In this case we still have to decide whether we found an *edge feature* or a *corner feature*. Let n_0 and n_1 be the normals spanning the maximum angle. If the feature is a corner, there is a normal n_k that is almost orthogonal to the plane spanned by n_0 and n_1, i.e. the angle between $(n_0 \times n_1)$ and n_k is small (cf. Fig. 10.6e). With

$$\varphi := \min_{k \neq 0,1} \angle(n_k, n_0 \times n_1)$$

and some corner threshold φ_{corner} we therefore classify the feature to be a corner if $\varphi \leq \varphi_{corner}$, and to be an edge otherwise.

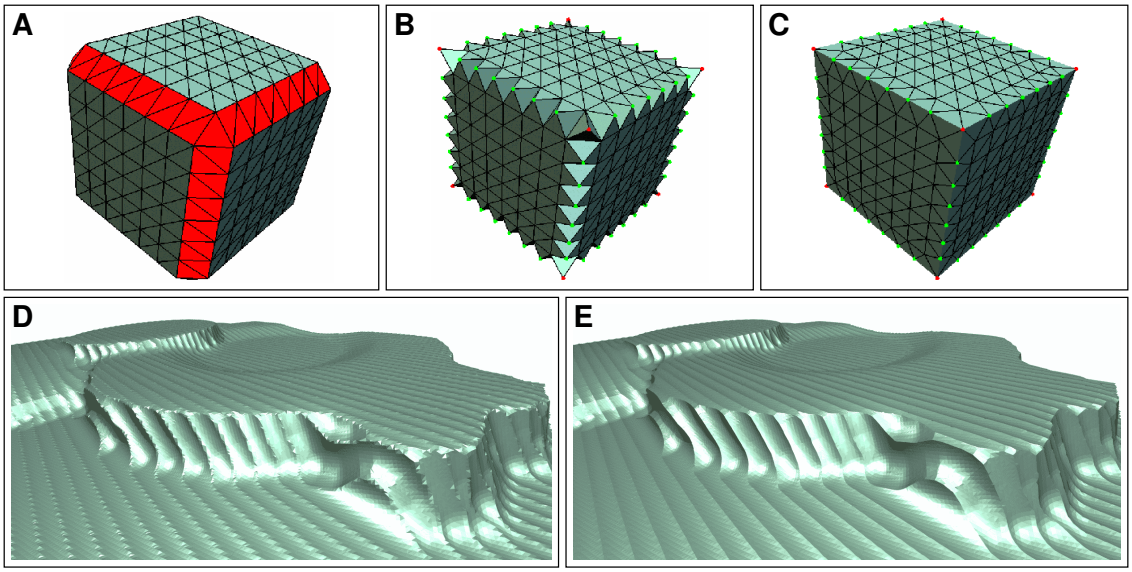


Figure 10.8: EMC: (a) feature detection, (b) feature sampling, (c) feature reconstruction, (d) milling example using MC, (e) same example using EMC.

Sampling In case we detected a feature within a cube we will find one additional sample point on that feature and include it into the resulting mesh. Each intersection point p_i and its surface normal n_i define a tangent plane. The sample point (x, y, z) should lie on the intersection of all these tangent planes (cf. Fig. 10.6f), leading to the linear system

$$\begin{pmatrix} n_0^T \\ \vdots \\ n_k^T \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} n_0^T p_0 \\ \vdots \\ n_k^T p_k \end{pmatrix}.$$

Note that this linear system might be under-determined as well as over-determined. In the case of an edge feature the different normals are samples from two parts of the surface, therefore we have two different normal directions. Since we intersect two tangent planes only, the solution is a line. The corresponding matrix then (numerically) has rank two only. For a corner feature there might be more than three distinct normal directions, resulting in a over-determined system.

Therefore, we do not try to find the exact intersection point, but approximate it by the point having the least quadratic distance from all planes, resulting in a Least Squares or Least Norm solution of the above linear system. This can most easily be achieved using the pseudo-inverse based on the *Singular Value Decomposition* (see [Golub and van Loan, 1996, Press et al., 2002] for further explanations).

Reconstruction We now construct the surface patch within the cube as we did in MC. If we did not detect a feature for a given cube the surface patch of the MC is used (based on the 256 case look-up-table).

If we detected a feature we have to include the computed sample point into the triangulation. The new vertex is connected to all edge intersection points using a triangle fan (cf. Fig. 10.6g). In case of edge samples this creates spikes in the resulting mesh. Therefore we need to flip edges crossing an edge-feature in order to reconstruct it correctly (cf. Fig. 10.6h). Fig. 10.8 shows the three steps — feature detection, sampling and reconstruction — for an example mesh.

Discussion

EMC consumes more memory, since it stores three directed distances instead of one. Due to the computations involved in feature detection and sampling it also requires approximately 20% – 40% more computation time than MC. However, the resulting mesh provides a much higher surface quality.

Note that this implies that for most of the cases EMC can achieve the same surface quality using a lower grid resolution than MC. Since the grid resolution strongly influences the running times, EMC might be faster in achieving the same quality level. In addition ϑ_{sharp} and φ_{corner} provide intuitive thresholds for the feature detection.

Chapter 11

Freeform Curves & Surfaces

Up to now the only explicit or parametric surface representation we discussed are polygonal meshes. Because of their conceptual simplicity and algorithmic efficiency we restricted to pure triangle meshes most of the time. However, if we want to construct high-quality smooth surfaces, they may not be the best choice. Due to its piecewise linear nature their curvature is either zero (inside a triangle) or infinite (across an edge or vertex).

For the modeling of mathematically well-defined sufficiently differentiable surfaces we therefore switch to polynomials of higher degree. We will start our description with freeform curves (univariate 3D functions) and extend the derived results to surfaces (bivariate 3D functions) afterwards.

In the same way we will finally extend from surfaces to (trivariate) volumetric functions and use these to perform freeform deformation of geometric objects.

This chapter will only present the basic facts about freeform curves and surfaces. This topic is part of the lecture Geometric Modeling, so the interested reader is referred to either this lecture, or the books [Farin, 1997, Prautzsch et al., 2002].

11.1 Freeform Curves

When considering parametric 3D curves we soon find out that polynomials are the preferable representation, since they have several advantages:

- Polynomials are C^∞ continuous.
- Evaluating polynomials requires additions and multiplications only, and hence is very efficient.

If we denote by Π^n the vector space of all polynomials of degree n, we will use parametric curves $f \in \Pi^n$ of the form

$$f : \mathbb{R} \to \mathbb{R}^3, \;\; t \mapsto \sum_{i=0}^{n} b_i F_i(t),$$

where the $b_i \in \mathbb{R}^3$ are vector valued coefficients (control points) and the $F_i(t) \in \Pi^n$ are scalar valued basis functions of the space Π^n.

A straightforward example for this representation is the *monomial basis* $F_i(t) = t^i$ that yields

$$f(t) = \sum_{i=0}^{n} b_i t^i.$$

Taking a closer look at this formula we realize that this is just a linear combination of the 3D coefficients b_i. From Sec. 2.1.2 we know that a sum of points has no geometric meaning unless it is an affine combination. Since the monomials in general do not sum to one, this representation for 3D curves lacks any geometric intuition: from the coefficients b_i we cannot guess the shape of the curve. As a consequence, we will look for a better suitable basis of Π^n.

11.1.1 Bézier Curves

The basis matching our needs is the *Bernstein basis*, built by the Bernstein polynomials

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i}.$$

Taking a closer look at these functions we can derive the following properties for them:

Partition of unity: The Bernstein polynomials sum to one:
$\sum_{i=0}^{n} B_i^n(t) \equiv 1$.

Non-negativity: The Bernstein polynomials are non-negative in the interval $[0, 1]$:
$B_i^n(t) \geq 0$ for $t \in [0, 1]$.

Maximum: The Bernstein polynomial $B_i^n(t)$ has its maximum at $t = \frac{i}{n}$.

Recursion: The Bernstein polynomials can easily be evaluated using a recursion formula:
$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t)$ using $B_0^0(t) \equiv 1$ and $B_{-1}^n(t) \equiv B_{n+1}^n(t) \equiv 0$.

The figure below shows the five Bernstein polynomials spanning the vector space Π^4.

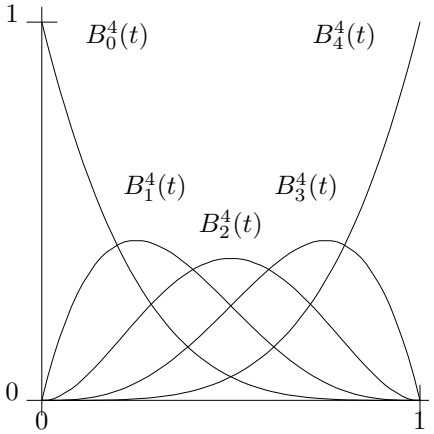


Figure 11.1: The five Bernstein polynomials of degree four.

Polynomial curves based on the Bernstein basis are called *Bézier curves* and have the form:

$$f : [0, 1] \to \mathbb{R}^3, \;\; t \mapsto \sum_{i=0}^{n} b_i B_i^n(t).$$

Transferring the properties of Bernstein polynomials one-by-one to Bézier curves results in the following advantages as compared to the monomial basis:

Geometric meaning: Since Bernstein polynomials build a partition of unity, evaluating a Bézier curve corresponds to an affine combination of the control points and therefore has a geometric meaning. As we will see, the control polygon $(b_0, \ldots, b_n)$ sketches the shape of the curve.

Affine invariance: Because evaluating Bézier curves is an affine combination, it is also invariant to affine transformations, i.e. in order to draw an affinely transformed curve one just has to transform the control points and evaluate the resulting curve.

Convex hull: The evaluation not only is an affine combination, but because of the non-negativity of the Bernstein polynomials it is also a convex combination of the control points. This has two implications: first, convex combinations are numerically very robust to evaluate, and second, the curve will lie in the convex hull of its control polygon.

Endpoint interpolation: When evaluating $B_i^n(t)$ at zero (or one) all Bernstein polynomials except the first (or last) one vanish, i.e. $B_i^n(0) = \delta_{i,0}$ and $B_i^n(1) = \delta_{i,n}$. Hence, the endpoints of a Bézier curve are given by the first and last control point:
$f(0) = b_0$ and $f(1) = b_n$.

Derivative: The derivative of a Bézier curve of degree n is a Bézier curve of degree $n-1$:
$f'(t) = n \sum_{i=0}^{n-1} (b_{i+1} - b_i) B_i^{n-1}(t)$.

Endpoint derivatives: Since the endpoint interpolation also holds for the derivatives, the endpoint derivatives at zero and one are given by the first two or last two control points, respectively: $f'(0) = n(b_1 - b_0)$ and $f'(1) = n(b_n - b_{n-1})$.

Local influence: Since $B_i^n(t)$ has its maximum at $\frac{i}{n}$, the influence of the control point b_i is concentrated around this parameter value. This is useful for curve editing, i.e. when control points are moved in order to modify the curve.

Exploiting the recursive definition of the Bernstein polynomials leads to a simple and very efficient evaluation algorithm for Bézier curves, the so-called *de Casteljau Algorithm*. Starting from the original control points $b_i^0 := b_i$, the following triangular array is built:

$$\begin{array}{ccccc} b_0^0 & & & & \\ b_1^0 & b_0^1 & & & \\ b_2^0 & b_1^1 & b_0^2 & & \\ \vdots & \vdots & & \ddots & \\ b_n^0 & b_{n-1}^1 & b_{n-2}^2 & \cdots & b_0^n \end{array}$$

Here, the points b_i^k are computed from points of the previous level by $b_i^k = (1-t)b_i^{k-1} + tb_{i+1}^{k-1}$, corresponding to the following update rule:

$$\begin{array}{ccc} b_i^{k-1} & & \\ & \overset{(1-t)}{\searrow} & \\ b_{i+1}^{k-1} & \xrightarrow{t} & b_i^k \end{array}$$

Since every point on level k is build by a linear interpolation of points on level $k-1$, the final point b_0^n is a polynomial of degree n in t. In fact it can be shown that the de Casteljau algorithm just makes use of the recursive definition of the Bernstein polynomials in order to evaluate a Bézier curve, i.e. the final point b_0^n eventually is the function value $f(t)$. This function value is computed by repeated convex combinations of control points, what is efficient as well as numerically robust. A geometric interpretation of this algorithm is depicted in Fig. 11.2.

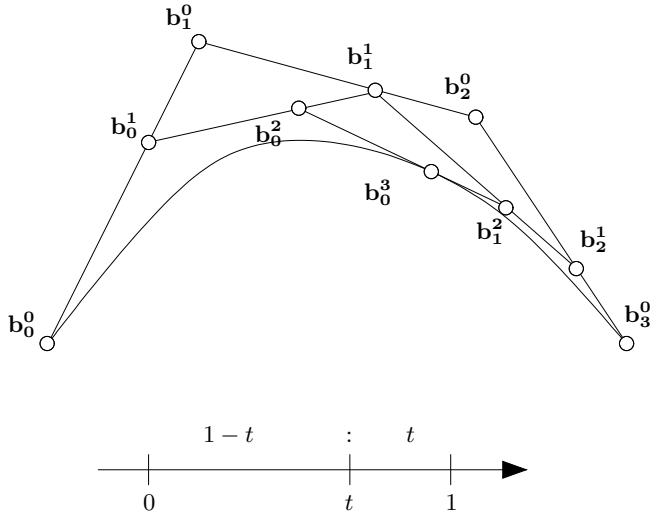


Figure 11.2: Geometric interpretation of the de Casteljau algorithm.

Additionally to function evaluation the de Casteljau algorithm can also be used to compute the tangent vector $f'(t)$. As mentioned above the derivative of a Bézier curve is

$$f'(t) = n \sum_{i=0}^{n-1} (b_{i+1} - b_i) B_i^{n-1}(t) = n \left(\sum_{i=0}^{n-1} b_{i+1} B_i^{n-1}(t) - \sum_{i=0}^{n-1} b_i B_i^{n-1}(t) \right).$$

This expression is just a difference of two Bézier curves of degree $n-1$, one defined by the control points $(b_0, \ldots, b_{n-1})$, the other defined by $(b_1, \ldots, b_n)$. Mapping this to the triangular scheme of the de Casteljau algorithm we see that the two respective function values are b_0^{n-1} and b_1^{n-1}. Hence, the last two points of the de Casteljau triangle provide us the tangent $f'(t)$ to the function value $f(t)$.

The de Casteljau algorithm can further be used to subdivide a given Bézier curve. Consider a curve $b(t)$ given by the control points $b_0, \ldots, b_n$, being parameterized over the interval $[0, 1]$. Splitting this curve at the parameter $x \in [0, 1]$ results in two curve segments $c(t)$ and $d(t)$, such that

$$c([0, 1]) = b([0, x]) \quad \text{and} \quad d([0, 1]) = b([x, 1]).$$

These curves can again be represented by Bézier curves of degree n, i.e. we have to find the corresponding control points $c_0, \ldots, c_n$ and $d_0, \ldots, d_n$. The de Casteljau algorithm provides us even with this information if we just evaluate the function $b(t)$ at the splitting parameter x. The control points are given by the upper and lower row in the triangular scheme, i.e.

$$c_i = b_0^i \quad \text{and} \quad d_i = b_i^{n-i}, \;\; i \in \{0, \ldots, n\}.$$

One subdivision step results in two new control polygons c_i and d_i which seem to better approximate the curve $b(t)$ than the original control polygon b_i (cf. Fig. 11.2). In general, repeating this curve subdivision k times results in 2^k control polygons. It can be shown that the polygon built by these control points converges quadratically to the original curve $b(t)$. This fact can be used to efficiently draw a Bézier curve: instead of evaluating the curve at 100 points and drawing the resulting 99 line segments, one can usually achieve the same visual quality by doing 4 subdivision steps (requiring only 11 de Casteljau runs) and drawing the resulting control polygons.

11.1.2 Spline Curves

One problem of polynomial curves is that the more degrees of freedom one needs the higher the polynomial degree has to be. E.g., interpolating n points requires a polynomial of degree $n - 1$.

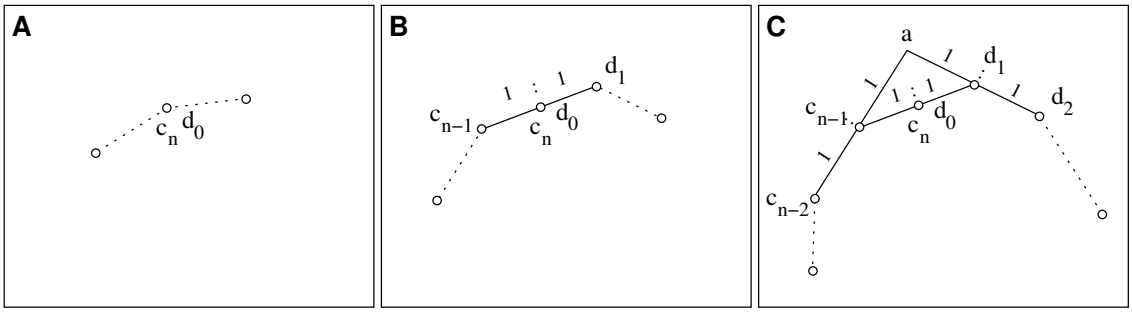


Figure 11.3: Joining Bézier curves: (a) C^0 junction, (b) C^1 junction, (c) C^2 junction (*A-frame*).

It is well known that polynomials of high degree tend to oscillate. The result will be a curve that satisfies all interpolation conditions, but also has an arbitrarily strange behavior between the interpolation points (e.g. it may form loops).

The solution to this problem is to use several (smoothly) connected polynomial curve segments $p_0(t), \ldots, p_k(t)$ to build one *piecewise polynomial* curve. The whole curve is defined over the parameter interval $[0, k+1]$, and each segment p_i is defined over $[i, i+1]$. If the connections between these segment are maximally smooth, this curve is called a *spline curve*. The formal definition of uniform splines of degree n is

$$\left\{ f \in C^{n-1} \ \wedge \ f|_{[i,i+1]} = p_i \in \Pi^n \right\},$$

i.e. each curve segment is a polynomial of degree n (therefore C^∞ continuous) and the whole curve (especially at the junctions) is C^{n-1} continuous:

$$\forall l \in \{0, \ldots, n-1\} \ : \ \frac{\partial^l}{\partial t^l} p_{i-1}(t) = \frac{\partial^l}{\partial t^l} p_i(t), \quad i \in \{1, \ldots, k\}.$$

Note that this is the maximum smoothness we can achieve using piecewise polynomials: if the segments would join with C^n smoothness, they would just be parts of *one* polynomial of degree n.

In practice we will mostly work on cubic polynomials that therefore have to be C^2 continuous at their junctions. We will now discuss how to construct two Bézier curves connected in a C^1 or C^2 manner. If these two curves are given by their control points $c_0, \ldots, c_n$ and $d_0, \ldots, d_n$, we derive from the endpoint-interpolation property, that the condition for a C^0 junction is simply

$$c_n = d_0,$$

(cf. Fig. 11.3a). From the last section we know that the endpoint derivatives of Bézier curves are determined by the first two or last two control points, respectively. Therefore, the condition for a C^1 junction gets

$$c_n = d_0 = \frac{c_{n-1} + d_1}{2},$$

meaning that the tangents $(c_n - c_{n-1})$ and $(d_1 - d_0)$ are parallel and equal in length (cf. Fig. 11.3b).

The second endpoint derivatives of a Bézier curve are analogously defined by the first three respectively the last three control points. It can be shown that for a C^2 junction these control points have to build a so-called *A-frame* together with a helper point a (cf. Fig. 11.3c).

If we now connect several cubic segments to a C^2 spline, we see that for all interior segments the complete set of control points is already defined by the continuity conditions. Hence, a cubic spline can be specified by the set of helper points a_i alone (cf. Fig. 11.4).

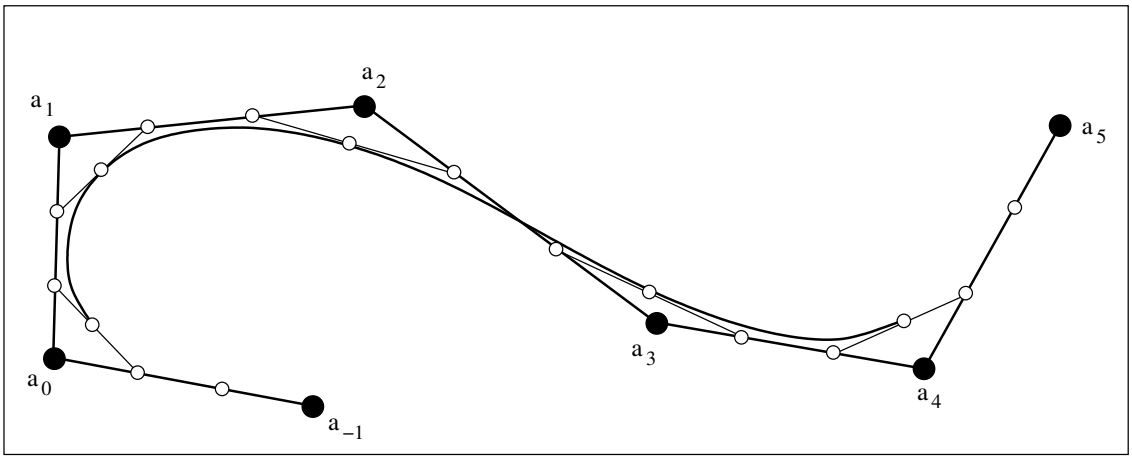


Figure 11.4: A cubic spline is defined by the A-frame helper points a_i alone.

Since these uniform cubic splines are just built from Bézier curves they share most of their properties. Some of their properties are:

- The Bézier points of the segments $p_0, \ldots, p_k$ are defined by $a_{-1}, \ldots, a_{k+2}$.
- Affine invariance
- Convex hull
- Maximal smoothness (C^2).
- Local control: moving one control point a_i just affects one Bézier segment.

11.2 Freeform Surfaces

For modeling 3D objects we have to use freeform surfaces instead of freeform curves. Now, after having set up all the mathematical framework in the last section, Bézier surfaces will turn out to be quite straightforward generalizations of the curve case. We will discuss two different generalizations, namely triangle patches and tensor-product patches.

11.2.1 Tensor-Product Patches

The basic idea for tensor-product patches is to define a surface by a curve whose points again move on curves, i.e. we have "curve-valued" curves. The area swept out by these moving curves defines the surface. If we consider a degree m Bézier curve

$$f(u) = \sum_{i=0}^{m} b_i B_i^m(u),$$

and move the control points b_i on curves of a second parameter v, we get

$$f(u,v) = \sum_{i=0}^{m} b_i(v) B_i^m(u).$$

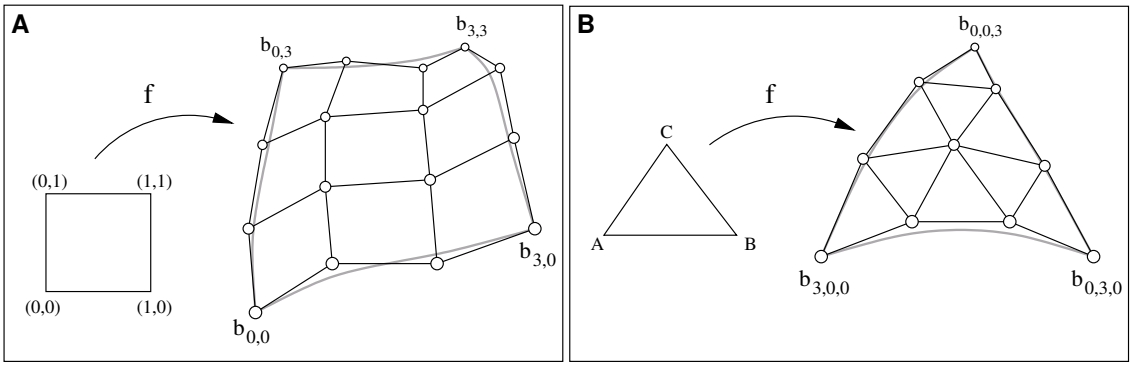


Figure 11.5: Bezier surfaces: (a) Tensor-Product Patches, (b) Triangle Patches.

If the curves we use for sweeping the control points are again Bézier curves (of a maybe different degree n), i.e. $b_i(v) = \sum_{j=0}^{n} b_{i,j} B_j^n(v)$ this finally evaluates to

$$f(u,v) = \sum_{i=0}^{m} \sum_{j=0}^{n} b_{i,j} B_j^n(v) B_i^m(u).$$

This so-called tensor-product patch is a bivariate function ($f : [0,1]^2 \to \mathbb{R}^3$) and is defined by a $(m+1) \times (n+1)$ grid of control points (cf. Fig. 11.5a).

Since tensor-product patches are just Bézier curves that move on other Bézier curves they inherit most of the properties we derived for these curves:

Affine Invariance: Evaluation is invariant to affine transformations.

Convex Hull: The surface lies in the convex hull of its control points.

Boundary Curves: The rectangular surface patch is bounded by the four Bézier curves $f(u,0)$, $f(u,1)$, $f(0,v)$, and $f(1,v)$.

de Casteljau: Tensor-product patches can be evaluated by using $m+1$ univariate degree n de Casteljau runs for the parameter v (yielding the control points $b_i(v)$), followed by another de Casteljau of degree m at parameter u, resulting in the function value $f(u,v)$.

By changing summation order one can also use $n+1$ runs at parameter u, followed by one step at parameter v. There is also a bivariate version of the de Casteljau that is handy when the patch is quadratic, i.e. $m = n$.

Derivatives: The partial derivatives of a tensor-product patch are

$$\begin{aligned} \frac{\partial}{\partial u} f(u,v) &= m \sum_{i=0}^{m-1} \sum_{j=0}^{n} (b_{i+1,j} - b_{i,j})\, B_i^{m-1}(u) B_j^n(v) \\ \frac{\partial}{\partial v} f(u,v) &= n \sum_{i=0}^{m} \sum_{j=0}^{n-1} (b_{i,j+1} - b_{i,j})\, B_i^m(u) B_j^{n-1}(v) \end{aligned}$$

The normal vector at a surface point $f(u,v)$ can be computed as $\frac{\partial}{\partial u} f(u,v) \times \frac{\partial}{\partial v} f(u,v)$.

Corner Interpolation: A tensor-product surface interpolates its four corner points, i.e. $f(0,0) = b_{0,0}$, $f(1,0) = b_{m,0}$, $f(0,1) = b_{0,n}$, and $f(1,1) = b_{m,n}$.

Corner Normals: The normal vectors at the four corner points are defined by the three control points meeting in that corner, e.g. the normal at $f(0,0)$ is parallel to $(b_{1,0} - b_{0,0}) \times (b_{0,1} - b_{0,0})$.

Smooth Junctions: If two tensor-product patches are to be joined along a boundary, the corresponding smoothness conditions are just the univariate conditions applied to each cross-boundary curve.

11.2.2 Triangle Patches

Bézier triangle patches are another generalization of Bézier curves which becomes more intuitive if we first rewrite the notation for Bézier curves. The parameter interval of a curve is the 2-simplex $[a, b] \subset \mathbb{R}$. The actual parameter value t can be represented by barycentric coordinates w.r.t. a and b:

$$t = \beta_a(t)a + \beta_b(t)b \quad \text{with} \quad \beta_a(t) + \beta_b(t) = 1.$$

Using this notation a Bézier curve can be written as

$$f(t) = \sum_{i+j=n} \frac{n!}{i!j!} \beta_a(t)^i \beta_b(t)^j b_{i,j}$$

where the control points are $b_{0,n}, \ldots, b_{n,0}$. While this notation looks more complicated than the previous one, it provides an easy way for the generalization to bivariate surfaces. In this case the parameter domain is a 3-simplex, i.e. a triangle $\Delta(a, b, c) \subset \mathbb{R}^2$. Any parameter value $u \in \mathbb{R}^2$ can again be specified by barycentric coordinates:

$$u = \beta_a(u)a + \beta_b(u)b + \beta_c(u)c \quad \text{with} \quad \beta_a(u) + \beta_b(u) + \beta_c(u) = 1.$$

From this we define a bivariate Bézier surface (a *triangle patch*) of degree n to be

$$f(u) = \sum_{i+j+k=n} \frac{n!}{i!j!k!} \beta_a(u)^i \beta_b(u)^j \beta_c(u)^k b_{i,j,k}.$$

Although the set of control points $b_{i,j,k}$ looks like a three-dimensional array, it is in fact a triangular arrangement since it is restricted by the condition $i + j + k = n$ (cf. Fig. 11.5b).

Again most of the univariate properties generalize to triangle patches:

Affine Invariance: Evaluation is invariant to affine transformations.

Convex Hull: The surface lies in the convex hull of its control points.

Boundary Curves: The triangular surface patch is bounded by three Bézier curves.

de Casteljau: The de Casteljau for triangle patches is basically the same like in the univariate case. For triangle patches we build a pyramidal scheme (instead of a triangular one) constructing a point on level k from three points on level $k-1$ by linear interpolation. The level $n-1$ consists of three points. These points define the tangent plane at the evaluated point.

Corner Interpolation: A triangle patch interpolates its three corners $b_{n,0,0}$, $b_{0,n,0}$, and $b_{0,0,n}$.

Corner Normals: The normals at the corners are defined by the three points meeting in that corner. The normal at $b_{n,0,0}$, e.g., is defined by the control points $b_{n,0,0}$, $b_{n-1,1,0}$ and $b_{n-1,0,1}$.

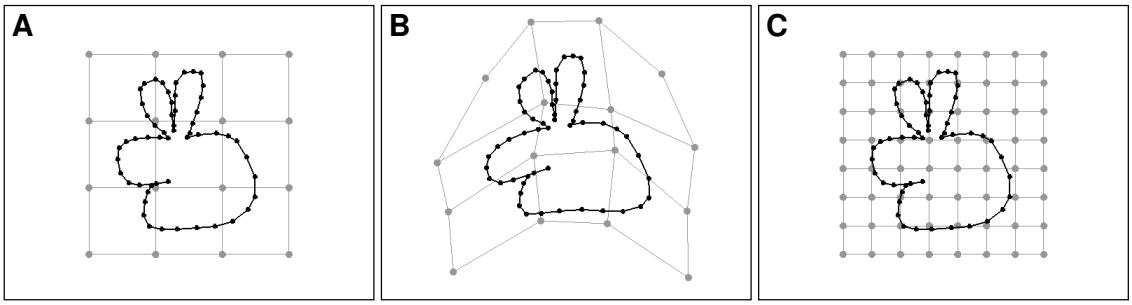


Figure 11.6: Freeform Deformation: Undeformed state (left). Moving control points results in deformation (center). Grid resolution control locality of modification (right).

11.3 Freeform Deformation

The freeform deformation approach (FFD) [Sederberg and Parry, 1986] deals with the modification or editing of surfaces. The goal is to develop an algorithm that provides a deformation method applicable to several kinds of geometry representations, like e.g. polygonal meshes or freeform curves and surfaces.

The solution to this problem is to consider the space around the object to be deformed. This can be imagined to be a parallelepiped of clear, flexible plastic in which the object is embedded. It can be specified by a lower-left corner X_0 and three (not normalized) orthogonal edges S, T, and U. This gives us a local coordinate system within the parallelepiped

$$X_0 + sS + tT + uU$$

such that the interior is specified by $0 \leq s, t, u \leq 1$.

The underlying idea is to deform this 3D space instead of the object itself. Each point X of the object has a certain position (s, t, u) w.r.t. the undeformed parallelepiped. Its deformed position X' should lie on the same relative position (s, t, u) w.r.t. to the deformed parallelepiped. Using this approach we are able to deform all objects that are defined by a set of 3D points, like e.g. polygonal meshes or freeform surfaces.

A function describing this space deformation has to be a map from $f : \mathbb{R}^3 \rightarrow \mathbb{R}^3$, such that $X' = f(X)$. The Bézier method has proven to be an intuitive and flexible approach to specify curves and surfaces (or polynomials in general), since the user can intuitively modify these objects by moving control points. Hence, the straightforward idea is to represent the trivariate space-deformation function in the Bézier basis.

This can be achieved by generalizing tensor-product patches one step further. Using the same arguments as in Sec. 11.2.1 we construct trivariate tensor-product volumes

$$f(s,t,u) = \sum_{i=0}^{l} \sum_{j=0}^{m} \sum_{k=0}^{n} b_{i,j,k} B_i^l(s) B_j^m(t) B_k^n(u)$$

which are now specified by a regular 3D grid of control points. The initial set of control points (the undeformed state) should represent the identity. This is the case for

$$b_{i,j,k} = X_0 + \frac{i}{l}S + \frac{j}{m}T + \frac{k}{n}U,$$

as can easily be derived from the univariate case (cf. Fig. 11.6a).

After this setup the freeform-deformation algorithm is very simple (cf. Fig. 11.6b):

1. The user moves control points in order to modify the trivariate deformation function:

$$b_{i,j,k} \mapsto b'_{i,j,k}$$

2. Each point X of the object's surface is transformed by inserting its local coordinates (s,t,u) (w.r.t. the undeformed parallelepiped) into the deformation function:

$$\begin{aligned} X' &= f(s,t,u) \\ &= \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} b'_{i,j,k} B_i^l(s) B_j^m(t) B_k^n(u) \end{aligned}$$

This method can now be applied to different kinds of geometry representation:

- Polygonal Meshes: Transform all vertices.
- Freeform Surfaces: Transform each control point. Although this is *not* the same as transforming each point of the surface (f is not affine in general) it is still preferable since it does not raise the polynomial degree.
- Implicit Functions: In order to deform implicits we have to apply the inverse transformation to the points we feed into the implicit function. Unfortunately, finding the inverse transformation f^{-1} is quite costly.

This approach also provides means to influence the *locality of deformation*, i.e. whether the deformation should be local or global. This can easily be done by adjusting the grid resolution: a coarse grid of few control points results in global modifications, a finer grid allows for local editing of small details (cf. Fig. 11.6c).

If only parts of the object are to be transformed, or if several neighboring parallelepipeds are to be used, we have to control the *smoothness of deformation*. Similar to the tensor-product surface case the k'th boundary derivatives $f^{(k)}$ of f are defined by the outer k layers of control points. Hence, leaving e.g. the outer two layers of control points untouched results in a C^1 continuous fading to the undeformed state at the boundary of the parallelepiped.

11.4 Inverse Freeform Deformation

The main drawback of freeform deformation is that one cannot prescribe that a certain point X on the object's surface should move by exactly d_X. This problem is addressed by the *inverse freeform deformation* approach: from a set of transformation constraints the control points are updated such that these constraints are met.

Suppose we are given the transformation constraints

$$f(p_c) \stackrel{!}{=} \overline{p}_c, \quad c \in \{0,\ldots,C\}.$$

If we re-enumerate terms in the function f like

$$\begin{aligned} f(s,t,u) &= \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n} b_{i,j,k} B_i^l(s) B_j^m(t) B_k^n(u) \\ &= \sum_{i=0}^{N} b_i \hat{B}_i(s,t,u), \end{aligned}$$

the transformation constraints result in the following linear system:

$$\begin{pmatrix} \hat{B}_0(p_0) & \cdots & \hat{B}_N(p_0) \\ \vdots & & \vdots \\ \hat{B}_0(p_C) & \cdots & \hat{B}_N(p_C) \end{pmatrix} \cdot \begin{pmatrix} b_0 \\ \vdots \\ b_N \end{pmatrix} = \begin{pmatrix} \overline{p}_0 \\ \vdots \\ \overline{p}_C \end{pmatrix}$$

Note that this in fact describes three linear systems since $b_i, \overline{p}_i \in \mathbb{R}^3$. However, the matrix is scalar valued and therefore has to be inverted only once. The problem is that this linear system may be over-determined as well as under-determined. While in the over-determined case we just compute the Least Squares solution to the problem, in the under-determined case we have to find another formulation such that the Least Norm solution makes sense (see [Press et al., 2002]).

If we consider differences of control points and surface points, i.e.

$$\Delta b_i := b_i' - b_i \quad \text{and} \quad \Delta p_c := \overline{p}_c - p_c$$

we get the modified linear system

$$\begin{pmatrix} \hat{B}_0(p_0) & \cdots & \hat{B}_N(p_0) \\ \vdots & & \vdots \\ \hat{B}_0(p_C) & \cdots & \hat{B}_N(p_C) \end{pmatrix} \cdot \begin{pmatrix} \Delta b_0 \\ \vdots \\ \Delta b_N \end{pmatrix} = \begin{pmatrix} \Delta p_0 \\ \vdots \\ \Delta p_C \end{pmatrix}.$$

Solving this system gives us the updates Δb_i we have to add to the original control points b_i in order to move surface points p_c by Δp_c, i.e. in order to satisfy the constraints. In this formulation the Least Norm solution uses the smallest amount of updates in order to meet the constraints, since the norm of the solution $\sqrt{\sum_i \|\Delta b_i\|^2}$ is minimal.

Part IV

Global Illumination

In the first part we have discussed the rendering pipeline as a means to render 3D geometry onto the screen. One of the reasons making this approach so fast is the fact that it works on local data only and therefore a pipelined approach can be used. As a consequence, the local rendering pipeline is nowadays efficiently implemented using hardware acceleration on modern graphic cards.

However, many illumination effects observed in reality are the result of global effects, i.e. effects on objects caused by other objects in the scene. In this part of the lecture we will discuss them.

One of the most substantial global effects is *visibility*. Visibility refers to the problem of determining which parts of a scene are visible to the viewer, and which are occluded by other objects; this problem is also referred to as *visibility determination* or *hidden surface removal*. It is obviously a global effect; considering a given object, we generally need to check whether it is occluded by any other object in the scene. Note that the rendering pipeline uses a global algorithm to resolve visibility as well, namely the *Z-buffer algorithm* (cf. Sec. 12.2.2).

Another global effect which is very important for visual realism are *shadows*. Since any object might cast a shadow onto any other object, computing shadows is also a global problem. Similar to visibility, solving this problem can also be included into the rendering pipeline by adding global pre-computation steps, multiple rendering passes and additional buffers.

There are various other global effects, such as indirect lighting, reflections and refractions of light in the scene. These effects inherently require the use of global illumination models, and cannot be realistically rendered using the rendering pipeline. There are specialized software algorithms which implement global illumination models, and we will discuss one of them in detail, namely *ray tracing*.

Chapter 12

Visibility

When rendering a scene containing many objects, it is clear that from a given viewing position some objects might be completely or partially hidden behind other objects. Obviously, in the final image we would like only those parts of objects to appear which are indeed visible. The problem of determining these parts is referred to as *visibility determination* (also known as *hidden surface removal*).

There are different techniques to address this problem:

- *Object-Precision Techniques*: These techniques check the object interdependencies in the scene to determine the parts of them hidden behind other objects. Therefore, their complexity is mainly determined by the complexity of the scene, i.e. by the number of objects or polygons in it.
- *Image-Precision Techniques*: Another way to solve the visibility problem is to consider the pixels of the final image. Finding the polygon visible through each pixel yields the color to fill the pixel with. Obviously, the complexity of this kind of algorithms is mainly determined by the resolution of the final image.
- *Mixed Techniques*: By combining both object and image precision techniques, and thus joining their advantages, one can find more efficient algorithms.

In the following sections we will discuss some of the most commonly used algorithms for visibility determination.

12.1 Object-Precision Techniques

Object-Precision techniques solve the problem of visibility determination in object space. By comparing objects against each other, they compute the parts of an object occluded by other objects and consider this information when rendering the objects.

The complexity of such algorithms is determined by the complexity of the scene, i.e. by the number of polygons contained in it. Since, in the worst case, each polygon might need to be checked against any other polygon in the scene, we get a complexity of

$$\mathcal{O}(\#polygons^2).$$

However, in general, not all pairs of objects have to be checked, and by using efficient algorithms we can usually expect a complexity closer to $\mathcal{O}(\#polygons \cdot \log(\#polygons))$.

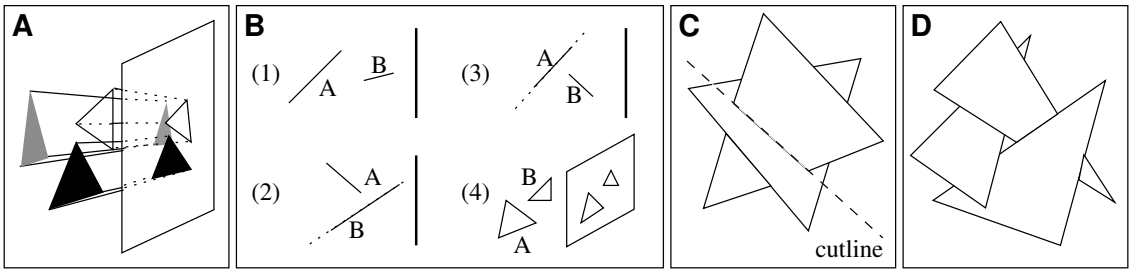


Figure 12.1: Painter's Algorithm: (a) idea (b) range cases (c) problem with intersecting polygons (d) cyclic overlap.

We will discuss the *painter's algorithm* here, which creates an ordering among the polygons such that rendering them in that order results in correct visibility. Since this is computationally very expensive, we will explain *Space Partitioning* techniques for faster visibility determination and *Culling* methods to accelerate rendering in general.

12.1.1 Painter's Algorithm

How would a painter proceed when painting an image of a scenery? Most probably, she would start by painting the background and continue to draw objects with decreasing distance to her. By doing so, nearer objects will be drawn on top of objects further away, occluding invisible parts (cf. Fig. 12.1a), and hence solving the problem of visibility determination.

The *painter's algorithm* follows this approach. In a first step, it sorts all polygons in the scene in decreasing distance from the viewer. Hence, polygons listed later in this ordering cannot be occluded by earlier polygons. Rendering the polygons according to that order then yields correct visibility, since nearer objects are drawn later and therefore overpaint more distant ones.

A good starting point is to sort all polygons w.r.t. the greatest z-coordinate z_{max} among all of their vertices. For the most cases this already suffices (cf. Fig. 12.1b1); but since polygons do not have just one z-value, but instead span a z-range $[z_{min}, z_{max}]$, we have to take a closer look at the cases where the z-ranges of polygons overlap.

So, let A and B be two polygons with overlapping depth-ranges,

$$z_{max}(A) > z_{max}(B) > z_{min}(A).$$

Note that in this scenario, A will be prior to B in the ordered list, a state to be either verified or corrected. There are some cases in which A can in fact safely be painted before B, implying that the list is indeed correctly ordered as it is:

- The $[x, y]$-ranges of the two polygons do not overlap.
- A is located behind the supporting plane of B[1] (cf. Fig. 12.1b2).
- B is located in front of the supporting plane of A (cf. Fig. 12.1b3).
- The projections of A and B onto the image plane do not overlap (cf. Fig. 12.1b4).

Conversely, there are some cases in which it might be safe to paint B before A:

- B lies behind the supporting plane of A.

[1]The supporting plane of a planar polygon is defined to be the plane it lies in.

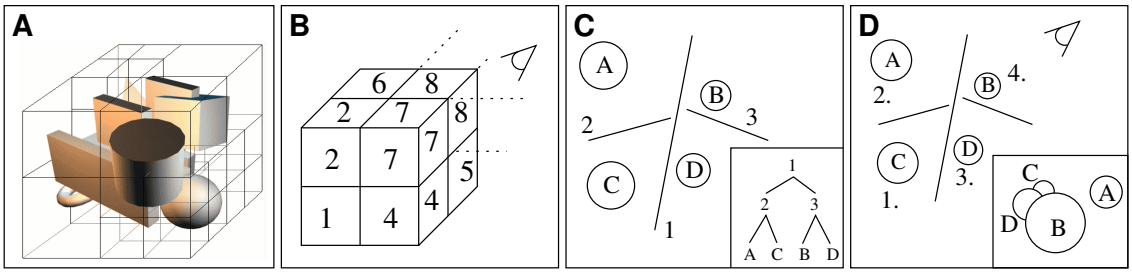


Figure 12.2: Space Partitioning: (a) octree (b) view-dependent order (c) BSP (d) BSP traversal.

- A lies in front of the supporting plane of B.

In the latter two cases we swap A and B in the list and proceed with checking the next ambiguous pair of polygons.

If all of the above tests fail, i.e. neither A nor B can be safely painted first, then we are in a situation similar to the one depicted in Fig. 12.1c. In this case, we need to split one of the polygons along the supporting plane of the other one. The three resulting polygons can always be painted in an order in which visibility will be correct.

Unfortunately, there are some special cases in which the algorithm still does not work. Consider a set of cyclicly overlapping polygons (cf. Fig. 12.1d). There is no order in which all of them can be rendered correctly. However, since any two of them can be painted in some order, the algorithm tries to sort them accordingly, and it gets trapped in an infinite loop within the cycle.

We therefore need to detect these cycles, and then split one of the involved polygons in the way already mentioned. After that split, there is an order in which these polygons can be painted with correct visibility, and the algorithm will find this order.

There are two main disadvantages of this algorithm, making it very inefficient for complex scenes. First, all polygons will be rendered, although most of them may be partly or fully occluded by other polygons. Also, the painting order is determined w.r.t. the z-values; if they change, e.g. when the viewer changes position, we need to reorder all polygons again. Sorting the objects for each frame is too complex for interactive applications, even for moderately complex scenes. In the following subsections we will explain two approaches to speed up this process: space partitioning and culling.

12.1.2 Space Partitioning

Space partitioning approaches try to accelerate the first step of the painter's algorithm, in which the polygons get sorted back-to-front according to their distance from the viewer.

As we will see, this sorting order can also be obtained by simply traversing special spatial data structures. Here the expensive part is the generation of these structures, they can be pre-computed for static scenes, however. Since the actual traversal is very efficient, using spatial data structures greatly improves the performance of visibility determination compared to the painter's algorithm.

Octrees

We already discussed octrees in section 10.2.1. An octree is a tree in which each node corresponds to an area of the scene, and in which each leaf references – in the scenario of space partitioning – a set of objects (cf. Fig. 12.2a).

The octree is built by recursively partitioning the bounding box around the scene into eight octants of equal size. The hierarchical structure of these subdivisions is represented by a tree structure: The bounding box is associated with the root node, and each node has its eight octants as children. The recursive subdivision stops when a node contains one object only, or when a maximum recursion depth has been reached (cf. Fig. 12.2a).

Note that since each leaf in an octree stores very few objects only, sorting them gets very easy. Hence, if we manage to find a back-to-front ordering for all octree cells, rendering the scene boils down to traversing the octree in this order and rendering the (few) objects contained in the leaf nodes using painter's algorithm.

Fortunately, due to the hierarchical structure of the octree, finding the order among all cells can be reduced to finding the order among the children of a node. Depending on the relative position of the viewer to the node, this problem can be solved by a simple table lookup (cf. Fig. 12.2b).

One disadvantage of octrees is that they are scene-independent: a node is always partitioned into its eight equally sized octants, regardless of how the objects might be distributed inside it. Thus, the resulting octree can get very imbalanced.

Binary Space Partition Trees (BSP)

BSPs are binary trees partitioning the objects in the scene. Each leaf is labelled with an object in the scene, and the parent of any two nodes references all objects associated with both of them. The root node hence obviously references all objects in the scene.

The partitioning itself is done according to a plane associated with each non-terminal node. This plane partitions the objects referenced by the node into two sets of objects (located on either side of the plane), which will then be referenced by the respective two child nodes (cf. Fig. 12.2c).

Note that there is no simple rule for how these partitioning planes should be chosen. In the easiest case, we always half the referenced part of the bounding box, leading to a scene-independent structure similar to an octree. Obviously, we should choose the planes according to the objects' distribution in the scene. That is, we choose planes such that the number of objects a child references is approximately half the number of objects its parent node references. This leads to a scene-dependent partitioning scheme. Although it is much more difficult to find such planes, we get a balanced tree of less depth, such that traversal is more efficient. Note that constructing the BSP for static scenes can be done off-line and will therefore not slow down the rendering process.

The traversal of the tree is very simple. Given a node and its bisecting plane, the viewing position will always be on one side of it. It is clear that objects on the distant side cannot occlude objects on the near side w.r.t. the eye, and hence can be painted first. Thus, we first traverse the subtree corresponding to the distant side of the plane and paint its objects, and then continue traversing the nearer objects (cf. Fig. 12.2d).

12.1.3 Culling

Culling tries to reduce the number of objects which are actually rendered. When rendering a complex scene it is clear that for most viewing positions not all objects in the scene will be visible. Therefore, discarding these invisible objects early in the rendering process can reduce the complexity of the rendered scene significantly, and saves computation time. Of course, this makes sense only if the invisibility tests themselves are simple.

Back-Face Culling

When considering a closed object, clearly polygons facing away from the viewer (*back-facing* polygons) are always occluded by those facing towards the viewer (*front-facing* polygons), as

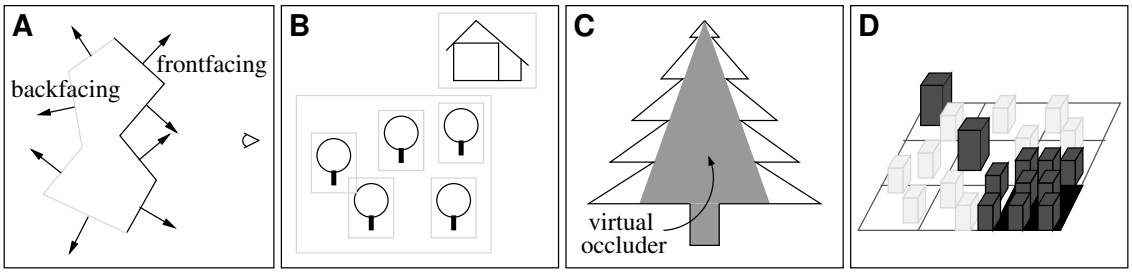


Figure 12.3: Culling. (a) back-face culling (b) bounding box hierarchies (c) virtual occluders (d) cell visibility.

depicted in Fig. 12.3a. Therefore, back-facing polygons do not have to be rendered at all, effectively halving the number of polygons to be processed.

Given the viewing position v and a polygon with normal n and barycenter p, this polygon is back-facing if normal and viewing direction both point into the same direction, i.e. if

$$n^T(p - v) > 0.$$

This can easier be detected by considering the orientation of the projected polygon, i.e. whether its vertices are ordered clockwise or counter-clockwise. Usually, faces are specified using a consistent vertex ordering. When they get projected onto the screen, back-facing polygons change their orientation, and can thus be distinguished from front-facing polygons, which keep orientation.

View Frustum Culling

The objects to be rendered are also determined by the view frustum; anything outside the frustum will not be visible from the current viewing position and hence can be discarded from the rendering process.

View frustum culling for polygons is automatically done by the clipping algorithms we already discussed in section 4.1. Note that these algorithms operate on polygon level, i.e. all the polygon vertices have to be transformed according to viewing position in order to clip. Because of this it would be much more efficient if we could perform this view frustum culling for whole objects at once. Bounding box hierarchies, discussed next, allow for even more efficient view frustum culling on several hierarchy levels.

Bounding Box Hierarchies

The idea behind bounding box hierarchies is to cluster objects into bounding boxes. Obviously, if the bounding box is not visible, then neither is any of the objects inside. Since the geometry of a bounding box is very simple, checking its visibility is much easier than performing visibility checks on each object inside the box.

By clustering bounding boxes into even larger bounding boxes, a hierarchy can be built, simplifying visibility checks even further. Note that this method provides for hierarchical view frustum culling. For example, in a landscape scenery with a forest, one could create a bounding box for each of the trees and then group these bounding boxes into a bounding box for the whole forest (cf. Fig. 12.3b).

Virtual Occluders

Similar to bounding boxes, virtual occluders are simple objects grouping together finer objects. However, the semantics are different. Objects are grouped together in a way such that from any

viewing position the objects always occupy at least the area the virtual occluder occupies, hence virtual occluders in effect approximate the inner contour of an object.

Using this fact, any object behind a virtual occluder does not need to be painted, since the objects inside the occluder will cover them. Again, it is easier to test objects against the simple geometry of a virtual occluder than to test it against each of the objects inside it.

Fig. 12.3c shows an example: assuming that the tree is dense enough to hide everything behind it in the core part, we can approximate it using a simple virtual occluder.

Cell Visibility

In large scenes, like e.g. a huge building or a city, several parts of the scene may not be visible from a given viewing position. In case of a building, for example, from outside a room none of the objects inside the room are visible, since they are occluded by the walls, assuming no window to be in the room and the doors to be closed. In a city, most of the buildings far away from the viewer (maybe except high ones) are usually occluded by buildings nearby.

We can exploit this fact by partitioning the scene into cells, and associating with each cell a list of objects which might be visible from inside the cell. Fig. 12.3d shows an example: the marked cell contains all objects inside the cell, some objects surrounding the cell which occlude objects further away, and some objects in other areas of the city which might be visible due to their height.

When rendering the scene, we only need to render the objects referenced by the current cell; all other objects can be omitted, since they will not be visible from within the cell. Note that this approach clusters viewpoints, in contrast to the other approaches, which cluster objects.

12.2 Image-Precision Techniques

Image-Precision techniques resolve visibility determination at pixel level of the final image. Here, the color of each pixel is determined by the foremost polygon. Ensuring that the pixels are filled by that color will produce correct results.

Obviously, the complexity of such algorithms is mainly determined by the resolution of the final image, i.e. by the number of pixels contained in it. However, for a given pixel we need to find the polygon visible through it, so the number of polygons also matters. Hence, we get a complexity of

$$\mathcal{O}(\#pixels \cdot \#polygons).$$

Note that finding the polygon is actually a search problem. Using efficient data structures and search algorithms, we can reduce the complexity to

$$\mathcal{O}(\#pixels \cdot \log(\#polygons)).$$

There are two algorithms we will discuss here. The first one is *Area subdivision*, which partitions the image plane until calculating visibility becomes trivial. The other algorithm, widely used nowadays, is the *Z-buffer* algorithm, which uses an additional buffer to keep track of pixel visibility.

12.2.1 Area Subdivision

The inherent problem in rendering the image is to find the polygons nearest to the image plane, since those will be the visible ones. In *Area Subdivision* approaches one tries to simplify the problem by subdividing the image area; here we will discuss *Warnock's Algorithm*.

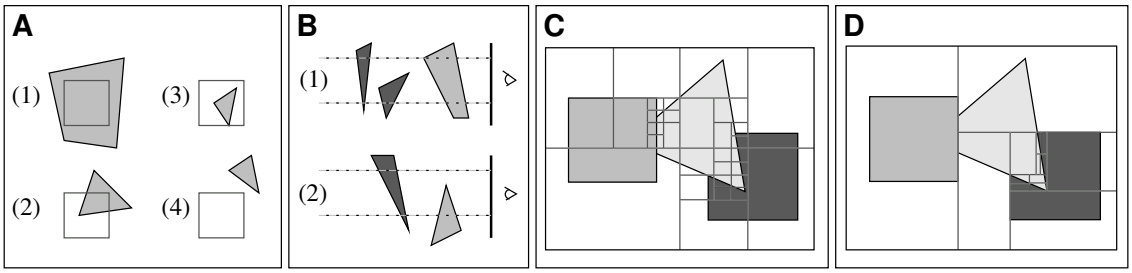


Figure 12.4: Area Subdivision: (a) interaction of a polygon with an area: cases 1 to 4 correspond to surrounding, intersecting, contained and outside, respectively. (b) occlusion test: 1 shows the easy case, 2 shows a case in which subdivision is advisable. (c) scene-independent subdivision, (d) scene-dependent subdivision.

In this algorithm, the image area is successively split into four equally sized sub-areas by inserting a horizontal and a vertical cut line, until either visibility is easy to resolve in the area, or a certain subdivision depth (up to pixel level) is reached. In the former case, the area is painted accordingly (see below), in the latter case, the color of the polygon nearest to the image plane in the middle of the pixel is chosen to fill it.

At this point, note that a polygon can interact with an area only in four different ways (cf. Fig. 12.4a): it may surround the area, intersect with it, be contained in it, or completely lie outside of it.

If at most one polygon interacts with a given area, visibility can easily be resolved:

- If all polygons are outside the area, it can be filled with background color.
- If there is only one polygon surrounding the area, it can be filled with the color of the polygon.
- If there is only one polygon in the area, either intersecting it or contained in it, we first fill the area with the background color and then paint the parts of the polygon inside.

If there is however more than one polygon involved, it could be the case that one of the polygons occludes all the others. In this case we can safely paint the area with that polygon's color. Otherwise, no decision can be made easily, and we subdivide the area further (cf. Fig. 12.4b).

This process is illustrated in Fig. 12.4c for some subdivision steps. Note that in most areas, visibility can be resolved very fast. Many subdivisions are only necessary at objects' boundaries; but even there, a maximum recursion limit is given by the pixel resolution.

We can accelerate subdivision using a scene-dependent approach by taking into consideration the orientation of the objects in the scene when cutting it. Instead of splitting the area into four equally sized sub-areas, we position the horizontal and vertical cut lines according to the objects, for example along their edges, or through their vertices. By doing so, the sub-areas are more likely to contain easy resolvable cases, and we need fewer subdivision steps (cf. Fig. 12.4d).

12.2.2 Z-Buffer

The Z-buffer is the simplest algorithm for resolving occlusions. The idea is to additionally create a buffer holding one depth-value per frame buffer pixel. During the rasterization stage, we incrementally fill this buffer and resolve visibility using it (cf. Alg. 10). Fig. 12.5a shows an example.

Before drawing any polygon, the Z-buffer is initialized with the depth value corresponding to the maximum z-value in the scene (i.e. the one of the view frustum's far plane), see below. Note that the order in which polygons are rendered does not matter in this algorithm.

Algorithm 10 Z-Buffer Algorithm.

```
for each polygon P
  for each pixel (x, y) in projection of P
    z = P's depth at (x, y)
    if (z ≤ zbuffer[x, y])
      zbuffer[x, y] = z
      framebuffer[x, y] = color(x, y)
```

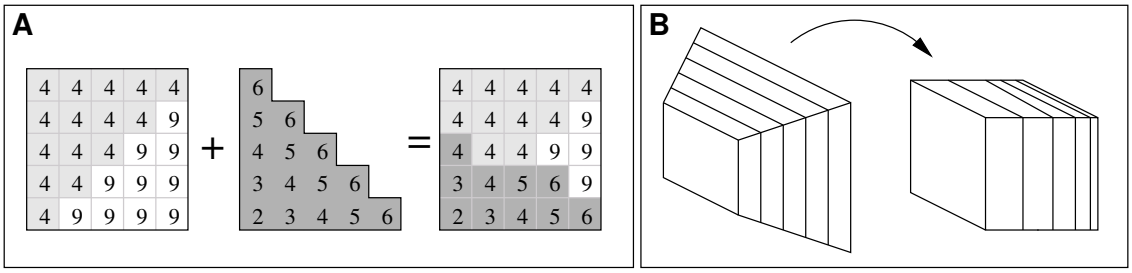


Figure 12.5: Z-Buffer Algorithm: (a) Z-buffer, (b) precision differences.

Apparently, the Z-buffer algorithm uses local data only (the current polygon). The only global structure is the Z-buffer itself. The algorithm is apparently linear, what looks like a contradiction to our assumption that the global visibility problem is at least of complexity $\mathcal{O}(\#polygons \cdot \log(\#polygons))$. This decrease in complexity is caused by the fact that we do not solve the exact visibility problem, but just a discretized version at pixel resolution.

Let us have a closer look at the z-values. As discussed in the first part of this lecture, we run through a set of transformations, until we actually draw the image. Some of these change the range of the z-values:

- Frustum Map: Points in the viewing frustum are mapped into the cube $[-1, 1]^3$.
- Window-to-Viewport Map: This corresponds to scaling the cube to the desired viewport. The depth component is scaled to the range $[0, 1]$.
- Discretization: The depth component is discretized according to the precision of the Z-buffer. If the Z-buffer can store m bits per pixel, we discretize to $\{0, ..., 2^m - 1\}$.

The Z-buffer holds the values obtained in the discretization step; in this step, a range $[z_i - \varepsilon, z_i + \varepsilon]$ will be mapped to one single value z_i. This is a problem inherent to discretization, which cannot be circumvented. There is a problem with the distribution of the z-values though. Since the frustum map involves perspective foreshortening, two equally large ranges

$$[z_1, z_2] \text{ and } [z_3, z_4], \quad \text{with} \quad z_1 < z_2 < z_3 < z_4 \text{ and } z_4 - z_3 = z_2 - z_1$$

are mapped to ranges of different extents

$$[z'_1, z'_2] \text{ and } [z'_3, z'_4], \quad \text{with} \quad z'_1 < z'_2 < z'_3 < z'_4 \text{ but } z'_4 - z'_3 < z'_2 - z'_1.$$

The difference depends on the relative positions of the z-ranges within the frustum; ranges near the far plane get much smaller than ranges at the near plane. This leads to a considerably high change in depth-resolution between near- and far plane in the normalized cube (cf. Fig. 12.5b).

Insufficient depth-buffer resolution might cause an effect called *Z-Fighting*: the depth resolution might not be high enough to resolve visibility correctly, hence polygons at different depths might be mapped to the same Z-buffer value in some regions, where they "fight" for visibility. Which

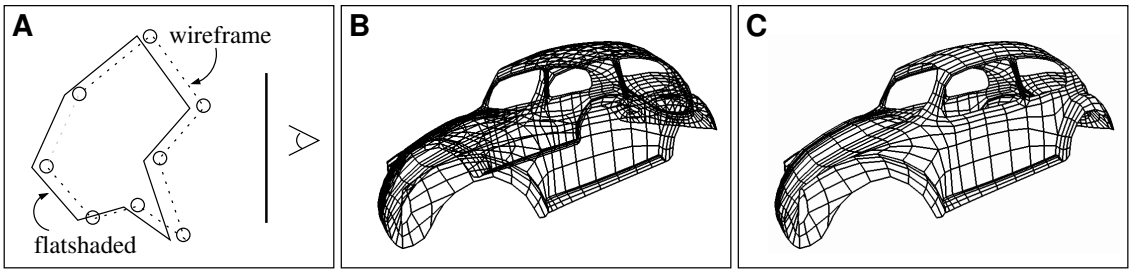


Figure 12.6: Hidden Line Removal: (a) method, (b) wireframe model, (c) hidden line removal.

one wins is in effect randomly chosen. Therefore, we get a high frequent noise or flickering when rendering the scene.

This is the reason why we should choose the smallest possible range between near- and far plane when setting up the frustum in order to achieve the highest possible depth resolution for the Z-buffer.

The simplicity of this algorithms is paid for by the cost of inefficiency. When rendering the scene, we will rasterize all polygons, although most pixels occupied by them will be overwritten later in the process. The worst case obviously occurs when the polygons are rendered back-to-front. Additionally, we need to compute a depth value for each pixel, what can however be included in the rasterization algorithm.

Another problem is the Z-buffer itself. Obviously, the buffer requires a lot of memory. However, since memory costs are constantly decreasing and since the Z-buffer algorithm is implemented in hardware on all modern graphics cards, using it is the fastest way to resolve occlusions.

12.2.3 Hidden Line Removal

In some applications, showing wireframe models might also be useful. Although wireframe models do not show lighting effects or colors, there are cases where it is useful to see the triangulation of the underlying surface. Unfortunately, in wireframe mode we see all the polygon edges even though some of them occluded by polygon faces, a fact making it much harder to grasp the object's geometry (cf. Fig. 12.6b).

An easy way to show only the ,,visible" edges of polygons is to use a two pass rendering approach: in a first step, we draw solid polygons filled with a constant color, e.g. the background color. This first rendering pass initializes the Z-buffer. After that, we draw the same model in wireframe-mode; however, we shift the model a little bit to the viewer's position in order to avoid Z-fighting (cf. Fig. 12.6a). Fig. 12.6b and Fig. 12.6c show the rendered image of a car model both in wireframe and hidden line mode.

12.3 Mixed Techniques

In this section we will present some hybrid techniques, which combine *object-precision* and *image-precision* techniques. The underlying idea is that the main difficulty of object-precision techniques is due to the fact that the scene might be very complex and that the various effects of the scene on the final image are hard to compute.

By only examining the effects of the scene in parts of the image it is possible to simplify the problem, leading to an overall performance improvement. Here, we will discuss the *scan-line approach* and *ray casting*.

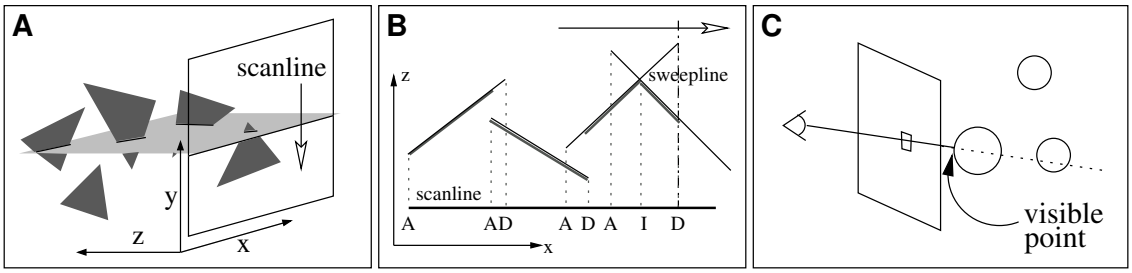


Figure 12.7: Scan-Line Approach and Ray-casting: (a) scan-line, (b) sweep line, (c) raycasting.

12.3.1 Scan-Line Approaches

The algorithm presented here is similar to the scan-line algorithm we used for rasterizing polygons, cf. Sec. 4.2.1. The idea is to reduce the dimension of the visibility problem by considering scan-line by scan-line.

In this case, each scan-line of the image corresponds to a plane through object space. For calculating visibility it suffices to consider the intersections of the polygons with the plane (cf. Fig. 12.7a). Within each of these planes we can solve the visibility problem by running through it with a sweep line and finding the positions where visibility changes (cf. Fig. 12.7b); this is a 1D problem.

The outer loop of the algorithm proceeds similar to the one described in section 4.2.1. We keep two lists of polygons, one containing the active polygons and one containing the passive ones. At the beginning, the passive list contains all polygons in the scene, and the active list is empty.

For each scan-line, we first check which of the passive polygons become active (when the scanline passes their minimum y-coordinate) and add them to the active list. Then, we check which of the active polygons need to be deactivated, what happens when the scanline passes their maximum y-coordinate. Then we proceed by calculating the visibility for the current scan-line.

To do this, we first intersect all polygons in the active list with the scan-line plane, leading to a set of intersection edges. Note that along a scanline visibility might only change due to the following events (cf. Fig. 12.7b) and we mark them for the sweep line:

- *Activation*: A new edge needs to be considered when its minimum x-coordinate is passed.
- *Deactivation*: An edge needs not to be considered anymore when the sweep line passes its maximum x-coordinate.
- *Intersection*: Visibility might also change whenever two (or more) edges intersect in a point.

Pay attention to the fact that in the x-ranges between these events the same edge stays visible. Therefore, we only need to check whether another polygon becomes visible at the marks, and can continue rendering the image using the corresponding polygon color until the next event is reached.

Apart from the fact that we have reduced the 3D visibility problem to a 1D problem, this algorithm exploits scan-line coherence. This applies to both the scan-lines and the sweep lines: in both cases we exploit the fact that very little changes occur from a visibility standpoint when jumping to the next line or pixel position.

12.3.2 Ray Casting

We already explained this algorithm when we discussed Volume Rendering (cf. Sec. 10.2.1). Here, we will summarize the algorithm under the aspect of visibility.

Ray casting calculates visibility on a per-pixel basis. That is, the algorithm runs through all image pixels and checks which polygon is visible through each of them. The corresponding color is then used to fill the pixel.

At a given pixel position, visibility is checked by sending a ray from the viewer's position through the pixel into the scene. There, the intersections of the ray with the scene's polygons are computed, and the nearest intersection corresponds to the visible polygon (cf. Fig. 12.7c).

The procedure can be accelerated by using *spatial data structures*, like those discussed in section 12.1.2. For example, the scene can be sorted into a BSP tree. Then, in order to compute the ray intersections, starting at the root node, we first traverse the subtree referencing the objects nearer to the viewer, and then traverse the other subtree. Note that this is the opposite order of the one we have used when utilizing BSP tree for the painter's Algorithm. Whenever we reach a leaf node, we intersect the ray with the objects associated with it; if there is an intersection, we are done (and the intersected polygon is the visible one), if there is no intersection, we continue the traversal.

Ray-Polygon Intersection

One problem we have not yet discussed is how to find out whether a ray intersects a polygon. Since we usually deal with triangles, we will restrict to ray-triangle intersections.

So, let $L_{p,q} = \{p + \lambda q \ : \ \lambda \in \mathbb{R}\}$ be the ray and $\triangle(a, b, c)$ $(a, b, c \in \mathbb{R}^3)$ the triangle in question. First, we need to compute the plane in which the triangle lies:

$$P(\triangle) = \left\{x \in \mathbb{R}^3 \ : \ n^T x = n^T a\right\}, \quad n = (b - a) \times (c - a).$$

Then, the intersection s of P and L is

$$s = p + \frac{n^T a - n^T p}{n^T q} \cdot q.$$

Now we only need to check whether s is inside the triangle by considering its barycentric coordinates $(\alpha, \beta, \gamma)^T$, which can be obtained by solving

$$\alpha a + \beta b + \gamma c = s.$$

If $\alpha, \beta, \gamma > 0$, s is inside the triangle, and we have found an intersection.

Chapter 13

Shadows

Another effect that can only be computed using global methods are *shadows*. In a scene, objects are illuminated by a given light source if there are no other objects between them and the light source. The not illuminated parts are said to lie in shadow, and the objects blocking the light (casting shadows) are called *occluders*. In this context the objects receiving shadows are called *occludees*.

Having discussed visibility in the last chapter, we realize that the computation of shadows is algorithmically similar. Instead of computing the visibility from the viewer's position, shadows can be found by computing visibility from the positions of light sources. Objects illuminated by a light source are obviously visible from it, and those parts of the scene not visible from a light source lie in shadow.

Let us take a closer look at what shadows are. Recall from section 2.3 that the light intensity on a surface point p with normal n as observed from viewing position v is according to the Phong model

$$C(p,n,v) = C_a + \sum_{\text{light } l} S(p,l) \cdot \left[C_d(p,n,l) + C_{sp}(p,n,l,v) \right].$$

In the above formula, C_a denote the ambient light, and C_d and C_{sp} denotes the diffuse and specular light, respectively. We now add a shadow term $S(p,l)$ to the formula, which takes into consideration that a light source affects the illumination of a point only if there are no objects blocking it (cf. Fig. 13.1a):

$$S(p,l) = \begin{cases} 0, & \text{if light from } l \text{ is blocked at point } p \\ 1, & \text{if point } p \text{ is visible from light source } l \end{cases}.$$

Note that the shadow term is binary because we are assuming all light sources to be infinitely small. However, if they would have a spatial extent, some points might be illuminated by only some parts of the light source, yielding a *soft shadow*, i.e. a shadow with decreasing intensity towards its boundaries. However, for the moment being, we only consider point light sources. Note that if there are several light sources in the scene, their shadows might join to so-called *deep shadows* (as opposed to *penumbrae* caused by the shadows of fewer light sources, cf. Fig. 13.1b).

All terms except the shadow term can be computed using local schemes. However, computing the shadow term is inherently global: for a given object's surface any other object in the scene (even itself) might contribute to this term.

Fortunately, there are algorithms which integrate this calculation into local rendering techniques, involving off-line pre-computation of the shadow's projection or the shadow's cone (based on global data). By using hardware acceleration for the computation of shadows based on this pre-computed data, the actual performance losses caused by multiple rendering passes are small, such

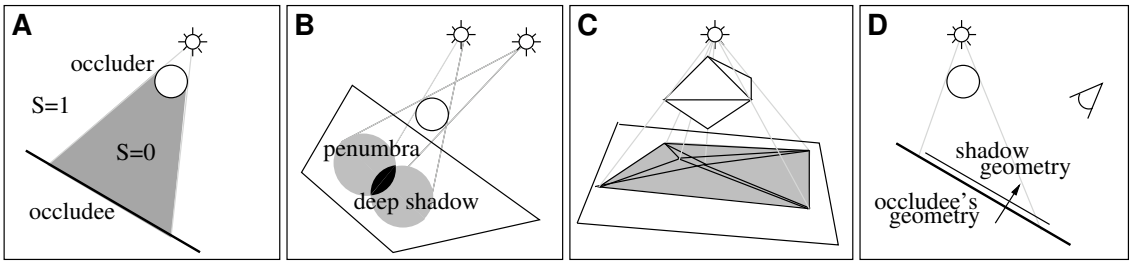


Figure 13.1: Shadows and Projected Geometry: (a) shadow term (b) penumbrae and deep shadows (c) projected geometry (d) shadow polygons.

that in most cases using the rendering pipeline to render shadows will be faster than using global illumination methods, which we will discuss in the next chapter.

The algorithms we present here can be classified according to the data they pre-compute, and the hardware units they use to accelerate rendering shadows:

	pre-compute shadow projection	pre-compute shadow cone
rasterization unit	projected geometry	shadow volumes
texturing unit	shadow textures	shadow maps

13.1 Projected Geometry

We have already seen that shadows are caused by objects blocking light. The regions on the occludee's surface which are in shadow appear darker than the rest of the surface. A very simple method to simulate shadows is therefore to simply add dark polygons to the scene which cover the regions in shadow. Then, the surface underneath will not be visible to the viewer, causing a visual effect similar to shadows.

As depicted in Fig. 13.1c, the shape of the shadow is determined by the projection of the occluder's geometry onto the occludee's surface. Assuming that the occludee is planar, this projection is similar to the projection we are computing when rendering the scene, the viewing position exchanged with the position of the light source, and the image plane replaced with the surface plane of the occludee. Having noted this, setting up the projection matrix should be easy.

Rendering the scene with shadows works in two passes. We first render the scene using standard illumination. Then, we multiply the shadow projection matrix by the modelview matrix, and render the occluders again. This time, however, we choose all polygons to have a dark shadow color. We can improve the visual results by making the objects transparent; then the occludee's surface underneath the shadow geometry will be partially visible.

Unfortunately, projecting the shadow geometry onto the occludee's surface causes Z-fighting artifacts (cf. Sec. 12.2.2). In order to avoid this we use a scheme similar to hidden line removal (cf. Sec. 12.2.3): we simply shift the projected occluders a bit into the direction of the viewer.

Of course, there are problems with this approach. Most apparently, we just simulate shadows. Accepting this for the sake of simplicity, also note that this scheme only works for planar occludees, although in a scene normally at most a few occludees (e.g. the ground) are planar. Even then, we need to generate shadow geometry for all such occludees, and for each light source, leading to a complexity bounded in $\mathcal{O}(\#lightsources \cdot \#occludees)$. Thus, if we have many occludees in the scene, or many light sources, the overhead caused by additional geometry might be substantial. Self shadowing, i.e. shadows cast by objects onto themselves, can also not be handled using projected geometry.

13.2 Shadow Textures

Another way of creating shadows is to use textures containing the shadow's image. This approach is very similar to *projected geometry*, but the shadows can be included into the geometry of the occludee, and no additional polygons are required.

As we have pointed out in section 13.1, a shadow on a surface is just the projection of the occluder's polygons onto that surface. If we had a texture with the shadow's projection colored black, and all other pixels colored white (a *shadow texture*), we could paint the shadow on the occludee by blending the shadow texture with the occludee's surface color.

Even better, by filtering the shadow texture with an averaging filter (cf. Sec. 15.2.1) we would smear the boundaries of the texture, thereby getting the impression of soft shadows.

But how can we get a shadow texture in the first place? As a matter of fact, we can use the rendering pipeline to perform this task. We simply put the image plane into the occludee's surface, and render all objects between the light source and the image plane from the position of the light source[1]. If we render all objects with black color, and use a white background, the resulting image will be the shadow texture.

For actually rendering the shadowed occludees we have to provide texture coordinates to get the correct shadowing results. To obtain these texture coordinates we just have to project the occludee's vertices onto the same image plane we used to render the shadow texture. The projected (x, y)-coordinates are the texture coordinates for that vertex.

Note that computing the shadow textures and texture coordinates needs to be done once for a given scene (view-independent), and therefore can be performed off-line, causing almost no speed penalty during the rendering process. Most important, shadow textures can be applied to non-planar occludees as well, what is not possible using projected geometry. However, apart from the fact that the shadow texture approach does not require additional geometry and can simulate soft shadows, it shares the disadvantages of projected geometry: it yields approximations to shadows only, and it does not support self-shadowing. As soon as light sources or occluders are moving the construction of textures has to be done for each frame, what, in general, will be too slow for real-time applications.

13.3 Shadow Volumes

In the previous section we have seen how to simulate shadows using a two-dimensional shape projected onto the occludees. In fact though, it would be more exact to compute the three-dimensional region that is in shadow w.r.t. a given light source and a given occluder [Crow, 1977].

This shadow region is called *shadow volume*: It is a cone-like volume, with the base having the shape of the occluders silhouette (as seen from the light source) and extending infinitely long on the far side of the occluder (cf. Fig. 13.2a).

How can we find the silhouette? Examining the occluder we can see that those of its edges build the silhouette, which separate front- and back-facing polygons (w.r.t. the light source). The side faces of the shadow volume are the extrusions of the silhouette's edges along the light rays intersecting the edge. Computing the shadow volumes has the complexity $\mathcal{O}(\#lightsources \cdot \#occluders)$; so it only makes sense to compute the shadow volumes if there are only few occluders in the scene.

Once we have computed the shadow volumes for a light source, it is clear that the corresponding light source has no effect on objects (or parts of objects) inside the shadow volume. Unfortunately, checking whether an object is inside the volume is generally very hard to accomplish. Luckily, ray casting (cf. Sec. 12.3.2) can be used to greatly simplify this task, and, as we will see, we can

[1] Alternatively, we could create one shadow texture per occluder

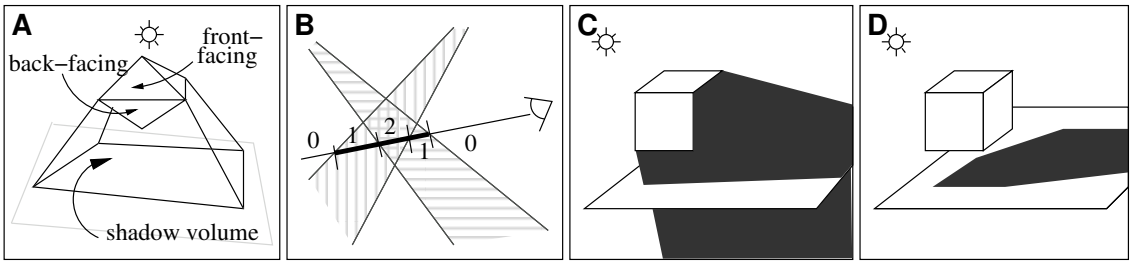


Figure 13.2: Shadow Volumes: (a) shadow volume (b) ray casting (c)(d) stencil buffers, marked regions correspond to positive values.

carry over the ideas used there to local rendering techniques by using multiple rendering passes and buffers.

Ray Casting Using Shadow Volumes

Recall that in ray casting we cast rays through each pixel of the final image, and for each ray we compute the first intersection with a polygon. The color used to fill the pixel is then determined by the color at the corresponding position of the intersected polygon.

In order to include shadows, we just need to find out which light sources affect that color. This is equivalent to identifying the shadow volumes in which the intersection point lies.

To check this we take into account the number of intersections of the ray with the shadow volumes before it reaches the intersection point (cf. Fig. 13.2b). Each intersection is either entering or leaving a shadow volume and we increment or decrement the counter accordingly. The intersection point lies inside of (at least) one shadow volume if the counter is non-zero.

Shadow Volumes in OpenGL

The process of counting the number of entry and leaving points can also be solved using stencil buffers and multiple rendering passes. Hence, we can use OpenGL (cf. Sec. 5), for example, to render shadow volumes, and we will describe in the remainder of this section how this can be done.

As already stated, we need multiple passes to render the final image. In a first pass, we render the scene using ambient light only. During this pass the Z-Buffer gets initialized, a fact that we will use later in the algorithm.

Now, consider the contributions of the light sources to the scene. They affect all pixels which do not lie in their shadows; thus, for each of the light sources, we need to find out which pixels are in their shadow (we will use the stencil buffer for this), and add the diffuse and specular contribution of the light source to all other pixels. More precisely, we perform the following steps for each light source:

1. We clear the stencil buffer. The frame buffer and Z-Buffer remain unaltered.

2. Now we render the *front-facing* polygons of all shadow volumes of the current light source. However, instead of updating the frame buffer, we just increase the corresponding stencil buffer entries by one. Note that we use the Z-Buffer for visibility tests while rendering, i.e. we only change the stencil if the Z-Buffer test has succeeded; we do not update the Z-Buffer though. In the stencil buffer we now find the number of entry points into shadow volumes for the pixel positions. Fig. 13.2c shows an example, where pixels marked in the stencil buffer after this pass are rendered black.

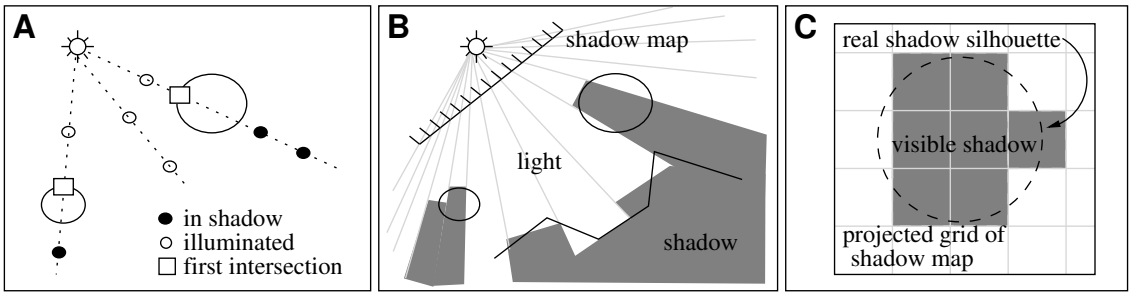


Figure 13.3: Shadow Maps: (a) shadows and distance, (b) shadow map visualization, (c) alias effects using shadow maps.

3. Next, we render the *back-facing* polygons in the same way, but instead of increasing the stencil buffer, we decrease it. This operation corresponds to counting the leaving points. Thus, afterwards only those pixels still have a positive stencil value which are inside of at least one shadow volume. Hence the stencil buffer represents the shadows of the current light source (cf. Fig. 13.2d).

4. In a last pass, we render the original scene, accumulating the result with the first pass into the frame buffer. This time, we add the diffuse and specular contributions of the current light source to the value already in the frame buffer. This update is restricted to pixels not marked in the stencil buffer, since they are the pixels illuminated by the current light source.

The main disadvantage of this algorithm is its speed: it requires the shadow volumes to be calculated. Note that the complexity of the shadow volumes depends on the complexity of the silhouettes, which in turn is determined by the occluders in the scene. These, however, can be arbitrarily complex. Luckily, shadow volumes are view-independent and can be computed off-line; but even assuming them to be available, they have to be rendered twice for each frame.

Apart from speed, shadow volumes have very nice properties. They are exact, i.e. they compute the boundaries of shadows exactly. Also, they support self-shadowing, and shadows on all kinds of objects, not just planar surfaces.

13.4 Shadow Maps

It can easily be verified that a point p is lit by a given light source, if there is no other surface point nearer to the light source than p itself (cf. Fig. 13.3a). If we could check this condition for a given point (or pixel), we were done.

Now, consider the Z-Buffer. For any given pixel, it contains values representing the distance between the viewer and the nearest surface seen through that pixel. By applying the modelview and projection matrix to any given point in the scene, we can map it to its corresponding Z-Buffer-value.

We therefore get the desired information by rendering the scene as seen from the position of the light source and storing the contents of the resulting Z-Buffer (cf. Fig. 13.3b). This Z-buffer image stores distances between light source and occluders and is referred to as the *Shadow Map* of the corresponding light source (cf. Fig. 13.3a) [Williams, 1978].

Using the shadow map, how can we find out whether a surface point p is lit by a light source L? This is simply a matter of using the combined modelview and projection matrix PM of the shadow map (the matrix that projects the points in the scene onto the shadow map plane). With

$$(x\ y\ z)^T := PM \cdot p,$$

we look up the entry $z_{min} = SM(x, y)$ at the shadow map position implied by (x, y). If $z \leq z_{min}$, p is lit by L, and we need to incorporate the diffuse and specular contribution of L when computing the pixel color for p. Otherwise, L has no effect on p.

Shadow Maps in OpenGL

Shadow maps are supported in OpenGL (cf. Sec. 13.4), and we can use them as follows:

1. We render the scene with ambient light. Doing so fills the Z-Buffer as well.
2. For each light source L we perform the following steps:
   (a) Render the scene using the shadow map of L and the shadow map extension of OpenGL. OpenGL uses this shadow map to find out which pixels are in shadow and marks them in the stencil buffer. Note that the result corresponds to the stencil buffer we have produced for shadow volumes (cf. Sec. 13.3).
   (b) Render the scene, adding the diffuse and specular contribution of L to all frame buffer pixels which are not marked in the stencil buffer.

Shadow maps, like shadow volumes, are a general method for computing shadows. All kinds of objects can cast and receive shadows (points, lines and polygons), and the algorithm can even handle *self shadowing*, i.e. cases in which an object casts a shadow onto itself. Furthermore, all steps in the algorithm can be done in hardware (at least on modern graphics cards), and therefore can be performed very fast. Note that shadow volumes need to be constructed without hardware support.

However, there is a very big disadvantage of this algorithm compared to shadow volumes. The shadow map is a discrete sampling of the shadow cones. This implies that the shadows we see in the scene have a discrete shape, and might not look realistic (cf. Fig. 13.3c and Fig. 13.5c). In the next subsection we will discuss *perspective shadow maps* weakening this effect; it cannot be completely prevented though. Note that shadow volumes do not have this problem, since they are an exact representation.

13.5 Perspective Shadow Maps

As already stated, shadow maps are discrete: they only sample depth values at the pixel positions of the map. Looking at the boundaries of the shadow cones, we get alias effects: a lookup in the shadow map might tell us we are in a shadow, although we are not, and vice versa (cf. Fig. 13.3b). This discretized shadow boundary gets visible on screen, if one shadow map pixel projects to many image pixels (*under-sampling*, cf. Fig. 13.4a).

Let us investigate how big shadow map pixels get when projecting them onto the image plane. Fig. 13.4b depicts the situation: we have a shadow map with pixel size h_s, and an image with pixel size h_i. For matters of simplicity we assume that the light source has distance one to the shadow map plane, as has the viewer from the image plane.

Consider the projection of one shadow map pixel onto some object in the scene. Clearly, the size of this projection is determined by the distance d_s between light source and object and by the angle α_s between the surface normal of the object and the projection direction. Putting them into context, we see that the size l_o of the projected shadow map pixel gets:

$$l_o = \frac{h_s \cdot d_s}{\cos \alpha_s}.$$

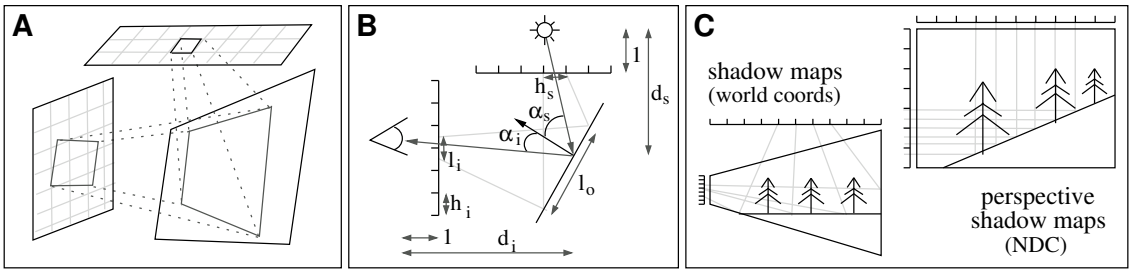


Figure 13.4: Perspective Shadow Maps: (a) problem, (b) calculation, (c) normal vs. perspective shadow maps.

We now have to project this shadow map pixel into the image plane, and again this depends on the angle α_i between viewing direction and the surface normal of the object and the distance d_i between object and viewing position. Thus, the size of a shadow map pixel on screen is

$$l_i = l_o \cdot \cos\alpha_i \cdot \frac{1}{d_i} = h_s \cdot \frac{d_s}{d_i} \cdot \frac{\cos\alpha_i}{\cos\alpha_s}.$$

Under-sampling occurs if $l_i > h_i$, i.e. if the shadow projection spans over more than one pixel. Therefore, our aim is to get l_i small. Take a closer look at the formula determining l_i. There are actually two aliasing effects we can observe:

- *Projective Aliasing*: This alias effect describes the size change due to the angle in which the projection rays hit the object's surface, i.e. by the ratio $\cos\alpha_i/\cos\alpha_s$. This factor depends on the actual objects in the scene, and we cannot control it.
- *Perspective Aliasing*: This alias effect is caused by the perspective projections involved, described by the term $d_s h_s/d_i$. We can obviously lessen this effect by increasing the resolution of the shadow map, thereby decreasing h_s. However, this is definitely not what we want to do. We therefore need to address the term d_s/d_i.

Keeping the shadow map plane at a constant position with a constant resolution and ignoring projective aliasing, the size of one pixel in the shadow map changes as a function of d_s and d_i. This is an implication of perspective foreshortening. We would like this to not happen: if the size of the pixel did not change according to the distances d_s and d_i we would have eliminated perspective aliasing.

Recall that in the rendering pipeline, perspective projections are obtained by mapping the viewing frustum to a cube, transforming world coordinates into normalized device coordinates (NDC). Using objects in NDC, rendering the image is then just a parallel projection.

The main idea of *perspective shadow maps* [Stamminger and Drettakis, 2002] is to specify the shadow maps not in terms of world coordinates, but in terms of NDC (cf. Fig. 13.4c). In this coordinate system, light rays as well as viewing rays are parallel, and therefore the area mapped to one single pixel in the shadow map and image remains constant. Therefore, perspective aliasing does not occur anymore; the only distortion is due to projective aliasing. Recall that projective aliasing is a property of the objects in the scene, and cannot be avoided in general.

Fig. 13.5 shows the impact of perspective shadow maps on the image. For both images, the same shadow map resolution is used. As you can see, with perspective shadow maps the shadow boundaries are smooth, in contrast to what is achieved with normal shadow maps.

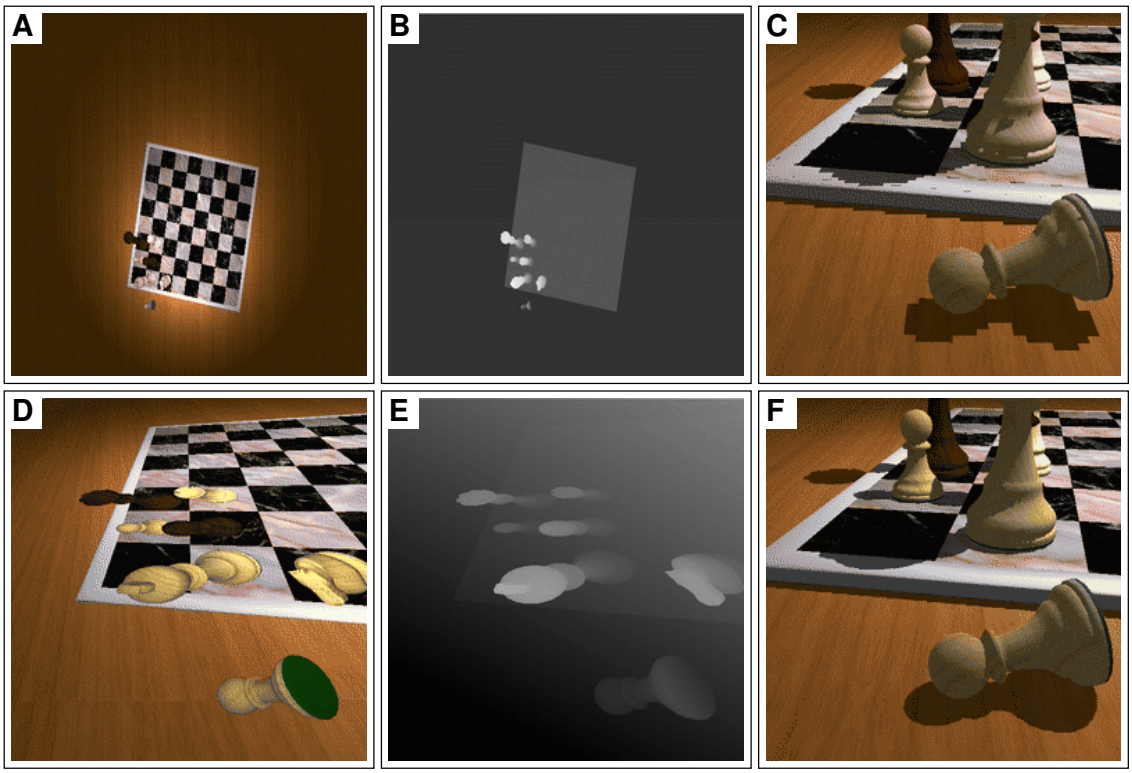


Figure 13.5: Shadow Maps vs. Perspective Shadow Maps: (a)-(c) normal shadow maps, (d)-(f) perspective shadow maps; (a)(d) view from light source, (b)(e) shadow map, (c)(f) zoom into image using shadow map.

Chapter 14

Ray Tracing

In the previous chapters we discussed two basic global effects, namely *visibility* and *shadows*. Both effects are vital for visual realism; fortunately, we could develop techniques to include them into the local rendering pipeline by adding global pre-computation steps and using additional buffers and multiple rendering passes.

The effects we can generate using the techniques discussed up to now have one thing in common: they consider the direct effects of light sources on surfaces only. Other global effects are caused by indirect lighting and reflections, and cannot be visualized realistically using the local rendering pipeline.

Indirect lighting is caused by the fact that light originating from a light source and hitting a surface is not absorbed completely, but to a certain amount also diffusely scatters further into the scene. Of course, any time light hits a surface, the reflected light will have less energy. Therefore, direct light, i.e. light directly emitted from a light source, has a much greater impact than indirect light; still, indirect lighting plays an important role for rendering highly realistic images.

Reflections are caused by light being specularly reflected (mirrored). As such, reflections can be considered a special case of indirect light; in contrast to indirect light though, the reflected light leaves the surface into one direction only.

In case an object is partially transparent, light might pass through it. Light transmitted in this way is referred to as *refractions*; although similar to reflections, refractions are harder to calculate. The problem is that refractions depend on the optical densities of the materials involved.

In *local illumination models*, indirect lighting is simulated by a constant ambient term. Although this might be a good approximation, for reasons that should have become clear indirect lighting largely depends on the positions and materials of the objects in the scene, and is by far not constant. Reflections and refractions are not considered by this model at all; however, by using environment maps (cf. Sec. 4.5.4) these effects can be simulated in the rendering pipeline.

Global illumination models try to account for all light in the scene affecting the image. There are three general approaches to this:

- *Backward Mapping*: We consider viewing rays from the eye through each pixel of the image and trace them back into object space in order to find the pixel colors. Ray tracing follows this approach.
- *Forward Mapping*: We consider the light emitted by the light sources, and trace it through the scene, until it finally reaches the eye. Radiosity uses this idea.
- *Hybrid Methods*: They combine forward and backward mapping. Photon mapping is an algorithm which uses this method.

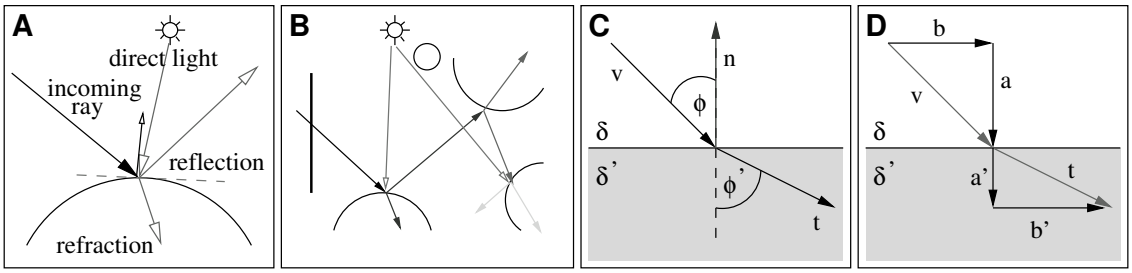


Figure 14.1: Ray Tracing: (a) light contributions, (b) recursive process, (c) law of refraction, (d) calculation of refracted vector.

Note that it is impossible to solve the global illumination problem exactly: this would require to trace rays into all directions, and furthermore to consider all outgoing rays when a ray hits a surface. Therefore, all methods discretize the process in one way or another. Due to this fact, not all of the algorithms can account for all illumination effects.

In this chapter we will mainly discuss ray tracing. Only in the last section we briefly summarize other techniques which will be discussed in detail in the lecture ,,Computer Graphics II“.

14.1 The Basic Ray Tracing Algorithm

The basic step in *Ray Tracing* [Glassner, 1993] is what we have already done in *ray casting* (cf. Sec. 10.2.1). For each pixel, we shoot a ray from the eye through that pixel into the scene. There, we calculate the first intersection of the ray with an object. The intersection point is the point visible at the given pixel position and it was illuminated using local lighting models.

In ray casting the point is illuminated using local lighting models; ray tracing extends this approach by also considering reflections and refractions.

Given a viewing ray intersecting a surface, we compute the directions from which light could have hit the surface in order to travel further along the viewing ray into the eye.

We cannot do this for diffuse reflections, since the incoming light is reflected equally into all directions, hence we would have to check for incoming light from all directions.

However, specularly reflected light must have had come from one specific direction (the reflected viewing ray), and the same holds for refracted light (the refracted viewing ray), giving us two new rays we need to take into account (cf. Fig. 14.1a).

Obviously, the considerations we made for the viewing ray are applicable on these two rays as well. The light coming from these directions thus depends on the next intersection with an object, more precisely on the direct light at this position and the light reflected and refracted there. This leads to a recursive procedure, illustrated in Fig. 14.1b. The resulting tree of rays between surfaces — the root of which is the viewing ray — is called the *ray tree*.

So, at each intersection of any of the rays with a surface, the color C at the intersection point (and hence the color being transmitted along the corresponding ray) consists of three components. Letting v be the direction of the incoming ray, p the intersection point, and n the surface normal at p, we obtain:

Direct Light C_{direct}: This is the light intensity caused by light coming directly from a light source and being reflected on the surface. We can evaluate this intensity using the Phong lighting model (cf. Sec. 2.3.1):

$$C_{direct} = C_{ambient} + \sum_{\text{lightsource } l} S(p,l) \cdot [C_{diffuse}(p,n,l) + C_{specular}(p,n,l,v)]$$

Algorithm 11 Recursive Raytracing algorithm.

```
Color trace(Point o, Direction v)
{
  if (p = find_intersection(o,v))
  {
    n = surface normal at p;
    r = reflection direction;
    t = refraction direction;
    C_direct = phong(p,n,v);
    C_reflect = α_reflect · trace(p,r);
    C_refract = α_refract · trace(p,t);
    return (C_direct + C_reflect + C_refract);
  }
  else return (0,0,0)
}
```

Note that we have added the shadow term $S(p,l)$ again. In ray tracing, this term can be evaluated by sending a ray from the surface point p to the light source l (*shadow ray*). If this ray intersects another object we have $S(p,l) = 0$, and $S(p,l) = 1$ otherwise.

Reflections $C_{reflect}$: This is the light specularly reflected at the surface, but coming from another surface rather than from a light source. Obviously, this light comes from the direction r of the reflected incoming ray v; we discussed how to calculate this direction in chapter 2.3.1: $r = v - 2nn^T v$.

The color $trace(p,r)$ observed at the first intersection of the reflection ray with another surface gives us $C_{reflect}$. Taking into account that the light energy attenuates due to reflections, we get:

$$C_{reflect} = \alpha_{reflect} \cdot trace(p,r)$$

Refractions $C_{refract}$: This is the light from another surface specularly refracted through the surface. We will discuss how to compute the refracted ray t in the next subsection. Analogously to reflections, the refracted ray contributes the following light intensity, using a damping factor corresponding to the surface's transparency:

$$C_{refract} = \alpha_{refract} \cdot trace(p,t)$$

Once we have computed all three contributions using recursive ray tracing, we add them together:

$$C = C_{direct} + C_{reflect} + C_{refract}$$

We stop the recursive tracing of reflected and refracted rays if the ray leaves the scene or a maximum recursion depth is reached. We could also associate with each ray its remaining light energy, and stop as soon as this energy falls below a threshold. This energy is the energy of the incoming ray damped by the corresponding weights $\alpha_{reflect}$ and $\alpha_{refract}$.

Alg. 11 shows a simplified version of the ray tracing algorithm.

14.1.1 Calculating the Refraction Ray

In order to calculate the refraction direction t, we have to use the *law of refraction* (also known as *Snell's law*). It states that

$$\frac{\sin\phi}{\sin\phi'} = \frac{\delta'}{\delta}, \quad \text{with } \phi = \angle(n,-v) \text{ and } \phi' = \angle(-n,t),$$

where n denotes the surface normal, v and t the incoming and the refracted direction, respectively, and δ and δ' the corresponding indices of refraction (optical densities) of the surrounding materials (cf. Fig. 14.1c). We can also assume that $\|v\| = \|t\|$.

First we decompose v into two orthogonal vectors, one being parallel to the surface normal n (cf. Fig. 14.1d):

$$v = a + b, \quad \text{with } a = nn^T v \text{ and } b = (I - nn^T)v.$$

Based on these vectors we want to find vectors a' and b' such that

$$t = a' + b', \quad \text{with } a' = \alpha a \text{ and } b' = \beta b.$$

Since $\|b\| = \sin\phi \cdot \|v\|$, we conclude from Snell's law

$$\begin{aligned} \|b'\| &= \sin\phi' \, \|t\| = \frac{\delta}{\delta'} \sin\phi \, \|t\| = \frac{\delta}{\delta'} \sin\phi \, \|v\| = \frac{\delta}{\delta'} \, \|b\| \\ \Rightarrow \quad \beta &= \frac{\delta}{\delta'}. \end{aligned}$$

With $\|v\| = \|t\|$ we can use the theorem of Pythagoras to obtain

$$\|t\|^2 = \|a'\|^2 + \|b'\|^2 = \alpha^2 \, \|a\|^2 + \left(\frac{\delta}{\delta'}\right)^2 \|b\|^2 \stackrel{!}{=} \|a\|^2 + \|b\|^2 = \|v\|^2$$

and solving this equation for α yields

$$\alpha = \sqrt{1 + \left(1 - \left(\frac{\delta}{\delta'}\right)^2\right) \cdot \frac{\|b\|^2}{\|a\|^2}}.$$

Summarizing all steps above we get a representation for t:

$$\begin{aligned} a &= nn^T v \\ b &= (I - nn^T)v \\ \alpha &= \sqrt{1 + \left(1 - (\delta/\delta')^2\right) \cdot \left(\|b\|^2 / \|a\|^2\right)} \\ \beta &= \delta/\delta' \\ t &= \alpha a + \beta b. \end{aligned}$$

In the special cases $\delta = \delta'$ (transmission of the light through the same material) or $\varphi = 0°$ (the ray hits the surface orthogonally) no refraction occurs, and the above formula correctly yields $t = v$.

14.2 Acceleration Techniques

The computational cost of ray tracing is dominated by the computation of ray intersections with objects in the scene. Note that we have to shoot first level rays for each pixel, and shadow rays, reflection rays and refraction rays at each intersection point, and all of these on several recursion levels.

We are therefore interested in simple tests to discard as many objects from further intersection tests as possible. Culling techniques, which we discussed already in the context of visibility (cf. Sec. 12.1.3) are very well suited for this task. Here, we will investigate the use of *bounding volumes* and *spatial structures* under the aspect of ray tracing, for which they are commonly used.

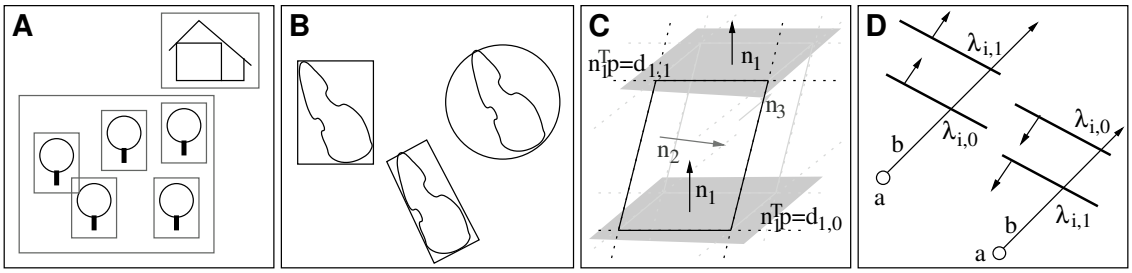


Figure 14.2: Acceleration Techniques: (a) bounding box hierarchy, (b) bounding volumes, (c) trapezoid, (d) entry and leaving points.

14.2.1 Bounding Volumes

One way to easily find out that a ray does *not* intersect a set of objects is to put them into a bounding volume. If the ray does not intersect the bounding volume, we know that it neither intersects any of the objects inside the volume. If the ray does intersect the bounding volume instead, we have to check for further intersections with the objects inside it.

Bounding volumes can be grouped hierarchically, i.e. several volumes can be grouped into larger bounding volumes, allowing for even faster hierarchical intersection tests (cf. Fig. 14.2a).

Note that it is not trivial to create a good hierarchy of bounding volumes. Too many volumes result in an overall increase of intersection operations, too few of them might lose potential performance improvements. Also, it is not obvious which objects to cluster together.

Besides the decision of which objects to group, there is also the question how the volumes should be shaped and aligned (cf. Fig. 14.2b). A prominent example of bounding volumes are bounding boxes. Aligning those to the coordinate axes will make it easy to test rays for intersections; however, large parts of the bounding box might be empty, leading to cases where the box is hit, but the objects inside are not.

Conversely, aligning the boxes to the objects will make it harder to calculate the intersections, but we will get a tighter fit, resulting in fewer "false intersections" with the box.

Another possibility is to use bounding spheres. Obviously, spheres do not have to be aligned, and ray-sphere intersections are easier to compute than those with general boxes.

Clustering the scenes in an optimal way is one of the current research topics, and no ideal algorithm has been found yet. Let us consider how to compute intersections between a ray and a sphere or box.

Ray-Box Intersection

The points inside a trapezoid B, the general form of a bounding box, can be written as

$$B = \left\{ p \in \mathbb{R}^3 \,:\, d_{1,0} \le n_1^T p \le d_{1,1} \;\wedge\; d_{2,0} \le n_2^T p \le d_{2,1} \;\wedge\; d_{3,0} \le n_3^T p \le d_{3,1} \right\},$$

with n_1, n_2, n_3 linearly independent and normalized. In the above formulation, any two inequalities with n_i, $1 \le i \le 3$, define the space between the two parallel planes corresponding to two opposite sides of the trapezoid. As depicted in Fig. 14.2c, the intersection of these three spaces builds a trapezoid. Note that we get a parallelepiped, if $\{n_1, n_2, n_3\}$ is a set of orthonormal vectors, and that this parallelepiped will be aligned to the coordinate axes if the n_i are the standard basis e_i.

Let the ray R be the set of points defined by

$$R(\lambda) = \left\{ a + \lambda b \,:\, \lambda \in \mathbb{R}^+ \right\},$$

where $a \in \mathbb{R}^3$ denotes the origin of the ray, and $b \in \mathbb{R}^3$ its direction.

To check whether R intersects B, we first find out for which values of λ the ray hits the bounding planes ($1 \leq i \leq 3$, $1 \leq j \leq 2$):

$$\begin{aligned} & n_i^T (a + \lambda b) \stackrel{!}{=} d_{i,j} \\ \Rightarrow \quad & \lambda = \frac{d_{i,j} - n_i^T a}{n_i^T b} \end{aligned}$$

Now we proceed similar to the Liang-Barsky algorithm for line clipping (cf. Sec. 3.2.2). For $1 \leq i \leq 3$, $\lambda_{i,0}$ and $\lambda_{i,1}$ are the ray parameters at which the ray enters and leaves the space between the corresponding set of bounding planes. Which of them is the entry parameter $\lambda_{i,in}$ and which the leaving point $\lambda_{i,out}$ depends on the orientation of the planes w.r.t. the ray direction (cf. Fig. 14.2d):

$$(\lambda_{i,in},\ \lambda_{i,out}) = \begin{cases} (\lambda_{i,0},\ \lambda_{i,1}), & \text{iff } n_i^T \cdot b > 0 \\ (\lambda_{i,1},\ \lambda_{i,0}), & \text{otherwise} \end{cases}$$

Using these points, R intersects B if

$$\max_{i=1,2,3} \{\lambda_{i,in}\} < \min_{i=1,2,3} \{\lambda_{i,out}\},$$

and in this case we need to check the objects inside the bounding box for further intersections.

Ray-Sphere Intersection

Recall from chapter 9.1.1 that the set of points on a sphere with midpoint $(m_x, m_y, m_z)^T$ and radius r can be written as

$$p^T \begin{pmatrix} 1 & 0 & 0 & -m_x \\ 0 & 1 & 0 & -m_y \\ 0 & 0 & 1 & -m_z \\ -m_x & -m_y & -m_z & \left(m_x^2 + m_y^2 + m_z^2 - r^2\right) \end{pmatrix} p = 0.$$

In the above equation, the points are specified in homogeneous coordinates. Therefore, we need to specify the ray in homogeneous coordinates as well, i.e. $R(\lambda) = a + \lambda b$ with $a, b \in \mathbb{R}^4$.

Now, the parameters for which the ray hits the surface of the sphere can be obtained by solving the following quadratic equation (note that $a^T Q b = b^T Q a$, since Q is symmetric):

$$\begin{aligned} & (a + \lambda b)^T \cdot Q \cdot (a + \lambda b) = 0 \\ \Leftrightarrow \quad & \underbrace{a^T Q a}_{=:C} + \lambda \underbrace{2 b^T Q a}_{=:B} + \lambda^2 \underbrace{b^T Q b}_{=:A} = 0 \end{aligned}$$

If solutions exist, they are

$$\lambda_{1,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}.$$

Actually, we are not really interested in the solutions of the equation. It suffices to check whether the sphere has an intersection with the ray. This is the case, if

$$B^2 - 4AC \geq 0.$$

Note that this intersection can be used for any bounding volume that can be expressed in terms of a quadric, e.g. an ellipsoid. Restricting to spheres results in slightly more efficient intersection computations though.

14.2.2 Spatial Subdivision

Another way to accelerate ray tracing is to use spatial data structures, for example one of the following:

- *Uniform Grid*: We can partition the scene into $l \cdot m \cdot n$ equally sized cubes. Each cell references the objects contained in it. Although only constant time is needed to access the cells, the main problem of this structure is the large number of cells in the grid, possibly leading to high memory requirements.
- *Octrees*: We can use an octree, as described in chapter 12.1.2. This is an adaptive scheme, since only non-empty cells are subdivided further. However, cells are always split into eight equally sized octants, regardless of the objects' positions, leading to an unbalanced tree.
- *BSP Trees*: BSP trees are a better way to subdivide the scene, and have been explained in chapter 12.1.2 as well. Subdividing the scene according to arbitrary planes makes it possible to select the two half-spaces in a way that each of them contains approximately the same number of objects. This results in a balanced tree and hence avoids high subdivision depth.

We can accelerate the ray-intersection using a uniform grid by traversing the cells in the order the ray intersects them. Finding the intersected cells along the ray corresponds to a 3D variant of the Bresenham algorithm (cf. Sec. 3.3). For each intersected cell we intersect the ray with all objects within it and stop as soon as we find the first intersection.

Using octrees or BSP trees, finding an intersection can be simplified by traversing this structure. Starting with the root node, we have to traverse the tree in increasing distance from the ray origin. In chapter 12.1.2 we described how to reference objects farthest-first for octrees and BSP; reversing this order results in a nearest-first traversal, as required here.

Before we descend down a node, we check whether the ray passes the area referenced by that node; if this is not the case we can discard checking its sub-nodes.

For each leaf we encounter we check the objects contained in it for intersection nearest-first. If an intersection occurs, we can stop the traversal. If the traversal completes without any intersection found, we can conclude that there is no intersection of the ray with the scene.

14.3 Super-Sampling

Up to now, we determine the color of a pixel by shooting a ray through it and assigning the color found at the first intersection.

This obviously leads to alias effects in high-frequent regions of the image, i.e. in regions, in which the color changes rapidly. For example, consider the boundary between two objects of different colors. Pixels along this boundary get one of these two colors, leading to alias artifacts (cf. Sec. 4.5.5). It would be much better to mix the color according to the contributions of the two colors in the region corresponding to that pixel.

One way to do this is to send multiple rays per pixel, and to average the resulting colors; this process is referred to as *super-sampling*.

We will discuss three types of super-sampling. *Uniform super-sampling* just increases the sampling density for the whole image, whereas *adaptive super-sampling* refines only parts of the image where high frequencies are detected. *Stochastic super-sampling* eliminates alias effects by shooting rays that are distributed randomly.

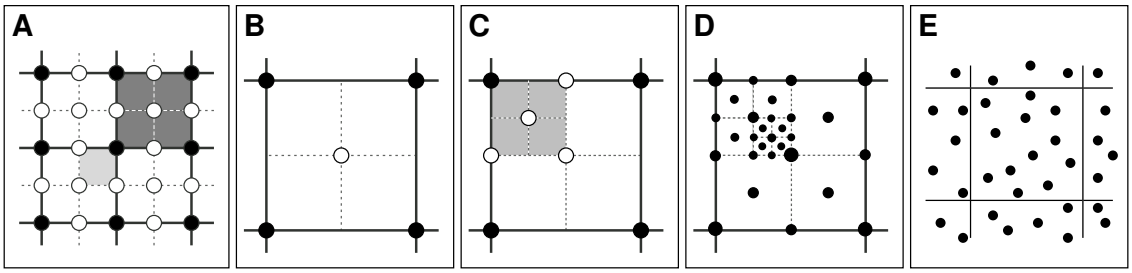


Figure 14.3: Super-sampling: (a) uniform super-sampling, (b)-(d) adaptive super-sampling, (e) stochastic super-sampling.

14.3.1 Uniform Super-Sampling

A very simple approach for super-sampling is to shoot more than one ray per pixel into the scene. Usually, one chooses a finer but still regular grid of samples, such as the one shown in Fig. 14.3a. Here, we use nine samples per pixel, resulting in a sub-pixel grid of doubled x and y resolution (since sample rays are shared, we actually compute four additional samples per pixel). A given pixel is then filled with the (possibly weighted) average of all sample colors in it.

In most regions of the image we can assume low frequencies; high frequencies normally only occur at object boundaries. This implies that in most parts of the image the higher sampling density is not required. In these regions, uniform super-sampling does not improve the image quality, but only wastes computation time.

Also note that with this method aliasing errors are not eliminated. We merely increase the resolution of the image, thereby making the errors less visible.

14.3.2 Adaptive Super-Sampling

The main problem of uniform super-sampling is its inefficiency. A high sampling density results in a substantial improvement of the image quality only in regions of high frequency.

Adaptive super-sampling chooses the sampling density based on the local frequencies. In homogeneous regions, where the frequency is low, the initial pixel grid is considered sufficient, whereas at object boundaries sampling density is locally increased.

In a first step we shoot rays through the four corners and the center of a given pixel (cf. Fig. 14.3b). If the resulting five colors differ just slightly, we are in a homogeneous regions and compute the pixel's color by averaging the five samples.

However, if the colors are sufficiently different, we split the pixel into four sub-pixels and recursively apply the adaptive super-sampling to each of them (cf. Fig. 14.3c). The recursion is stopped if a sub-pixel is considered homogeneous or if a maximum recursion depth is reached. Fig. 14.3d shows an example for this process.

14.3.3 Stochastic Sampling

Since we cannot refine our grid infinitely and since the resulting refined grid is also a regular sampling pattern, we still get alias effects in the image using uniform or adaptive refinement.

A way to eliminate alias effects is *stochastic super-sampling*. Here, we have a number of samples we can use per pixel, but instead of placing them on a regular grid we randomly scatter them over the area of the pixel (cf. Fig. 14.3e). Of course, the samples should get equally distributed in order to get a representative sampling of the pixel.

The irregularity of this approach effectively avoids alias effects. However, this method adds noise to the image. But since noise is actually more pleasant to the eye than alias effects, this is a trade off we might be willing to accept.

14.4 Discussion and Outlook

Using the acceleration techniques we discussed, ray tracing can be implemented very efficiently. Since ray tracing can also be parallelized easily, it is nowadays possible to render images at interactive frame rates. The resulting images provide a much higher visual quality than those created using the local rendering pipeline, making ray tracing more and more competitive.

The complexity of ray tracing is logarithmic in the number of objects, while the complexity of the rendering pipeline is linear in it. Hence, from a certain scene complexity on ray tracing will be more efficient than the rendering pipeline.

The results we achieve using ray tracing is not perfect though (cf. Fig. 14.4a). Note the following:

- Indirect lighting is not considered. From the global light terms we only consider reflections and refractions of direct light, and still use an ambient light level to simulate indirect diffuse lighting within the scene.
- Light sources are assumed to have point shape. Therefore, we do not get soft shadows as we would using area light sources (cf. Fig. 14.4b).
- Caustics will not appear (cf. Fig. 14.4c). These are effects caused by light being bundled by reflections or refractions, resulting in bright spots. This effect cannot be modelled by a backward-mapping algorithm like ray tracing.

Radiosity

Radiosity follows another approach. It considers the global light exchange within the scene by calculating a balance of the light energy emitted by all surfaces in the scene. For that approach to be feasible the surfaces get discretized into patches. Light energy is injected into the scene by the light sources, so the computations trace the light from the light sources into the scene (*forward mapping*).

Radiosity calculates a light intensity for each surface patch in the scene. This approach considers diffuse lighting only, i.e. in contrast to ray tracing, radiosity cannot model reflections and refractions. However, it can handle area light sources and soft shadows.

Another consequence is that radiosity is view-independent. Once the global light balance has been computed, we can use it to render any given view of the scene.

We will discuss the radiosity method in detail in the lecture „Computer Graphics II“.

Combined Ray Tracing and Radiosity

The previous paragraphs should have made clear that ray tracing and radiosity complement each other. Ray tracing mainly computes reflections and refractions of light, whereas radiosity computes diffuse and indirect lighting. By adding the results of the two approaches we can improve the visual realism further.

Note that we still do not get all effects. Caustics, for example, can be handled by neither ray tracing nor radiosity, and therefore cannot be handled by just combining their results.

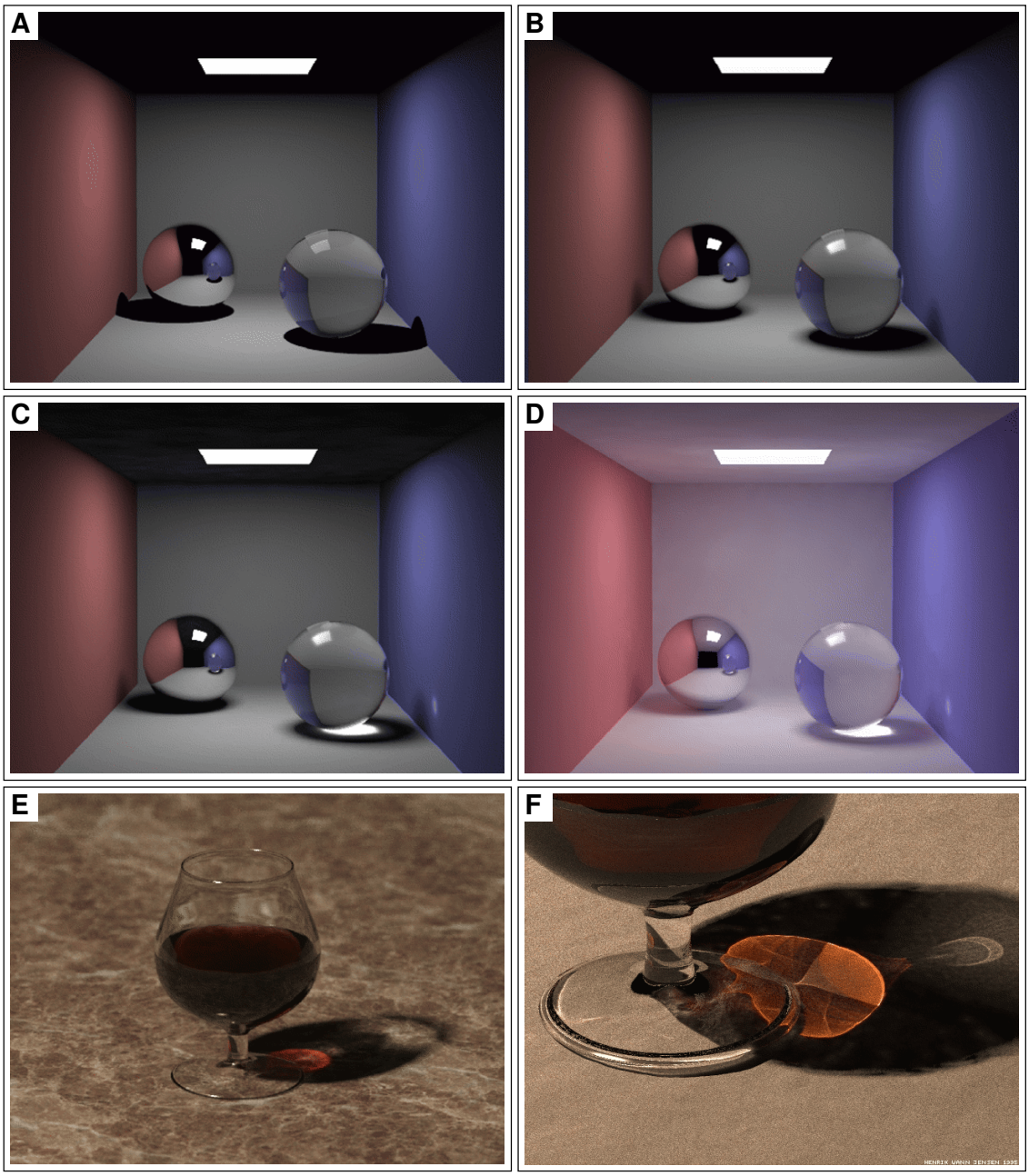


Figure 14.4: Global Illumination Rendering: (a) Ray Tracing solution, (b) plus area light sources causing soft shadows, (c) plus caustics due to the transparent sphere, (d) plus indirect diffuse lighting, (e) photo realistic rendering of a cognac glass, (f) closeup of the caustics.

There are advanced techniques, such as *photon mapping*, which can visualize these effects, but we will defer their discussion to ,,Computer Graphics II“ as well. Just for you to get a glimpse of how much photo realism is possible nowadays, take a look at Fig. 14.4d-f.

Part V

Images

We discussed several geometry representations and how to render them using local and global illumination, finally resulting in an image. In this last part of the lecture we will further process these images and thereby conclude the discussion of the whole image-generation pipeline.

An image can be considered to be a function mapping pixel positions in the image to a color (cf. Sec. 2.2.2):

$$f: \begin{array}{ccc} \mathbb{R}^2 & \longrightarrow & \mathbb{R}^3 \\ (u,v) & \longmapsto & (r,g,b) \end{array}$$

Of course, this would be just the mathematical representation of an image; in reality, we have to discretize this function.

On the one hand, the domain of the function is discretized to a regular grid of pixels (*domain discretization*). If the image is w pixels wide and h pixels high, the function gets

$$f: \begin{array}{ccc} \{0,...,w-1\} \times \{0,...,h-1\} & \longrightarrow & \mathbb{R}^3 \\ (u,v) & \longmapsto & (r,g,b) \end{array} .$$

The range of the function, i.e. the color space, has to be discretized as well (*range discretization*). In the simplest case, one would just restrict the number of distinct values each of the color channels can take (*quantization*). If we allow n values for each of the channels, we get

$$f: \begin{array}{ccc} \{0,...,w-1\} \times \{0,...,h-1\} & \longrightarrow & \{0,...,n-1\}^3 \\ (u,v) & \longmapsto & (r,g,b) \end{array} .$$

One of the topics in this part will be how to choose a *quantization* or *color table* such that the reduction of image quality will be minimal. In this context, we will also discuss techniques to give the impression of more colors where actually much fewer are available (e.g. B/W printers).

However, we will first discuss two types of image transformations, namely *filtering* (e.g. removing noise) and *geometric transformations* (e.g. scaling or rotating images).

Finally, we will talk about image compression techniques trying to represent an image using a minimal amount of data. In this context, we will also discuss progressive transmission of images.

Chapter 15

Image Transformations

Transforming images, i.e. applying filters or geometric transformations, is one of the most important operations on images. We will discuss several types of *filters* here, which can reduce noise in images, smoothen or sharpen them, improve their color spectrum, and even find contours in the image.

As we will see, filters can be evaluated and designed by investigating their effect on the frequency spectrum of a signal. For this reason, we will introduce the *Fourier transform*, a transformation that maps a function to its frequency spectrum.

We will also revise the topic of *aliasing* here; however, this time focussing on the reasons and on general solutions to this problem.

A completely different type of transformations are *geometric transformations*, like e.g. scaling or rotating images. We already discussed these transformations for the 3D case and will adept them to images here.

15.1 Fourier Transform

When analyzing signals or designing filters it is very helpful to examine their behavior by looking at their frequency spectrum, i.e. determining which frequencies exist to which amount in the signal.

We can try to represent a given function by a sum of sine waves having different frequencies, amplitudes and phase shifts. Fig. 15.1a depicts how a rectangle signal can be composed from (infinitely) many sine waves. Finding this set of waves is called *Fourier analysis* and yields the representation of the given function in the *frequency domain*.

Conversely, we can compose a function from a given set of frequencies by just summing up the corresponding waves (*Fourier synthesis*), mapping the representation from frequency domain to *spatial domain*.

15.1.1 Definition

These two tasks are performed by the *Fourier transform* and the *inverse Fourier transform*, respectively. The Fourier transform $F(u)$ of a given function $f(x)$ is

$$F(u) = \int_{-\infty}^{\infty} f(x) \cdot e^{-i2\pi ux}\, dx, \tag{15.1}$$

where $i := \sqrt{-1}$ is the complex imaginary unit. This transform maps $f(x)$ from spatial domain to $F(u)$ in frequency domain. $F(u)$ reveals to which amount the frequency u is contained in the signal $f(x)$. Since $F(u)$ is a complex number, it represents phase shift ϕ as well as amplitude $|F(u)|$ by its polar coordinates. We will denote this transformation by symbol

$$f(x) \circ\!\!-\!\!\bullet F(u).$$

In a very similar way, the inverse Fourier transform reconstructs a signal from a given frequency spectrum:

$$f(x) = \int_{-\infty}^{\infty} F(u) \cdot e^{i2\pi ux}\, du, \tag{15.2}$$

and we denote this map by

$$F(u) \bullet\!\!-\!\!\circ f(x).$$

15.1.2 Discussion

In order to understand these two transforms we first step back from signals to the 3D Euclidean vector space. Given any vector p and an orthonormal basis b_i, we can determine how much of a certain b_i is contained in p by orthogonally projecting p onto b_i

$$P_i := \langle p, b_i \rangle .$$

Using these coefficients P_i, we can represent p w.r.t. the vectors b_i, corresponding to a basis change:

$$p = \sum_i P_i \cdot b_i.$$

The same very basic technique from linear algebra is the mathematical background of the Fourier transform. The signals $f(x)$ are integrable functions of the vector space L^2, the dot product of which is

$$\langle f, g \rangle := \int_{-\infty}^{\infty} f(x)\overline{g(x)}\, dx.$$

A complex wave of frequency u is given by

$$e^{i2\pi ux} = \cos(2\pi ux) - i\sin(2\pi ux) =: e_u,$$

so the amount a frequency u is contained in the signal $f(x)$ simply gets

$$\langle f, e_u \rangle = \int_{-\infty}^{\infty} f(x) e^{-i2\pi ux}\, dx = F(u).$$

The infinite set of waves $\{e_u, u \in \mathbb{R}\}$ builds a basis of the function space L^2, therefore we can represent the signal $f(x)$ w.r.t. this basis by

$$f(x) = \int_{-\infty}^{\infty} \langle f, e_u \rangle \cdot e_u\, du = \int_{-\infty}^{\infty} F(u) \cdot e^{i2\pi ux}\, du.$$

Fig. 15.1b shows the Fourier transforms of some example functions. Let us take a closer look at the first example, showing an oscillation around 0.5. In the Fourier transformed we can observe one peak at zero, corresponding to the constant offset 0.5 (representing the frequency $e_0 \equiv 1$). This value is referred to as the *direct current* of the signal. The frequency of the oscillation causes the other peaks at ± 0.25. Note that the magnitude is axis-symmetric, since negating the frequency corresponds to inverting the phase angle, leaving the magnitude unchanged.

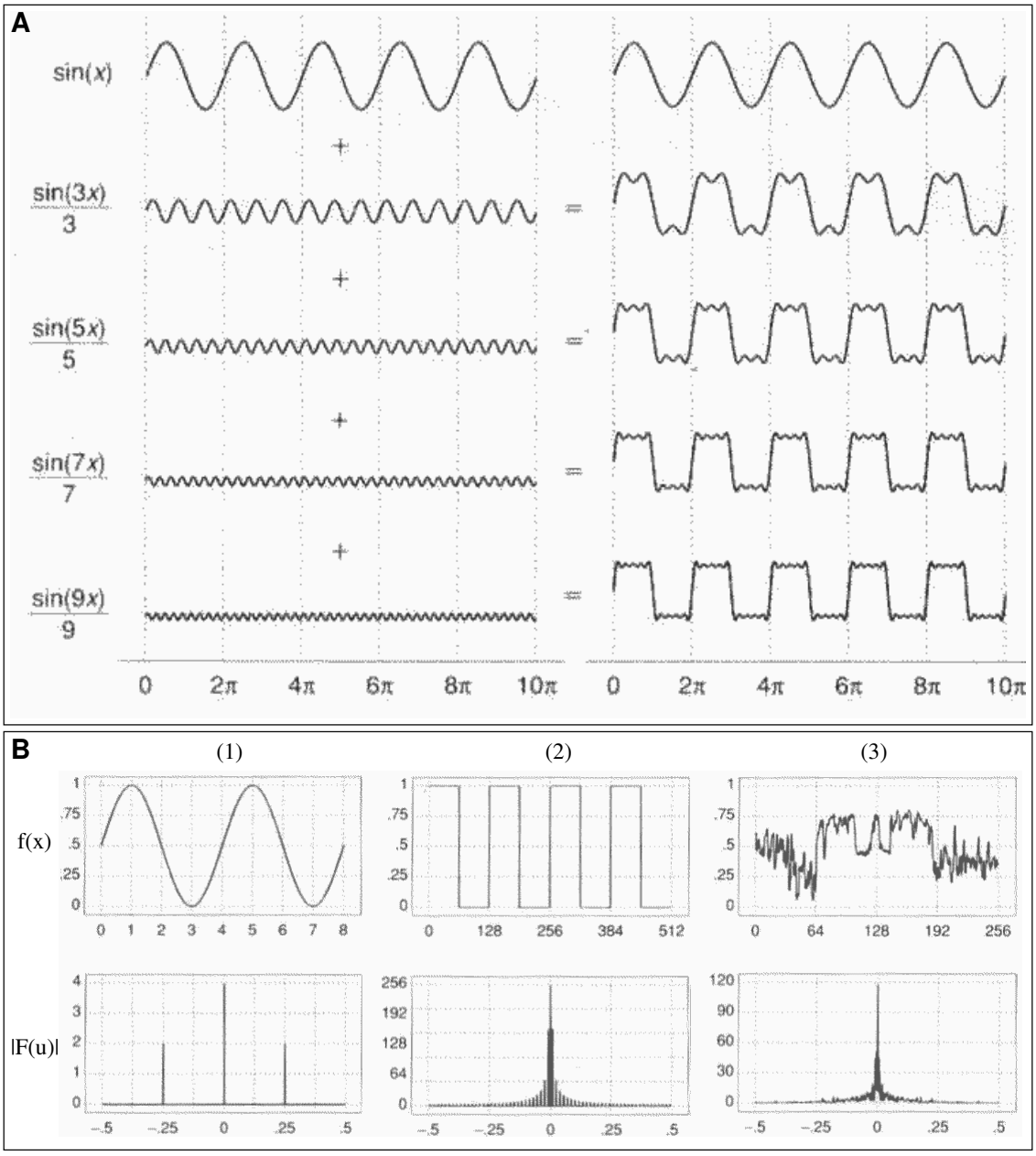


Figure 15.1: Fourier Transform: (a) Composition of a rectangle signal by sine waves, (b) Examples of the Fourier transform.

15.1.3 Convolution Theorem

The convolution of two functions f and g is defined to be

$$(f * g)(x) := \int_{-\infty}^{\infty} f(t)g(x-t)\,dt. \tag{15.3}$$

The so-called *convolution theorem* states that

$$(f * g)(x) \quad \circ\!\!-\!\!\bullet \quad F(u) \cdot G(u) \tag{15.4}$$

$$\text{and} \quad (F * G)(u) \quad \bullet\!\!-\!\!\circ \quad f(x) \cdot g(x). \tag{15.5}$$

Hence, convolution in spatial domain corresponds to multiplication in frequency domain. This correspondence explains what actually happens when convolving f and g: the frequencies of f are modulated with the frequencies of g. Convolution can therefore be regarded as the application of a frequency filter on f, with G (and thus g) determining its characteristics. For this reason, g is called the *characteristic function* or *transfer function*, since it amplifies or attenuates the frequencies of f.

Exploiting this fact, it is easy to derive a specific transfer function $g(x)$ in the spatial domain. We just create a suitable frequency modulation $G(u)$ in the frequency domain, and transform it back into the spatial domain.

15.1.4 Discrete Fourier Transform

The images we want to process are not continuous functions, but discrete samplings instead. Therefore we have to generalize the Fourier transform to discrete signals, resulting in the *Discrete Fourier Transform*. For reasons of simplicity, we first handle one-dimensional discrete functions defined over a finite range:

$$f = [f_0, f_1, ..., f_{n-1}].$$

As we will see in a minute, it is advisable to periodically continue discrete functions, since this has some advantages when applying the Fourier analysis.

Before discussing discrete signals, let us have a look at periodic continuous ones, i.e. let f be a periodic function with period T. Now consider the different frequencies contained in f. Since f has period T, the lowest strictly positive frequency in f must be $f_T = 1/T$ (called the *fundamental frequency*). All other frequencies contained in f must be integer multiples of this fundamental frequency, since otherwise f would not be periodic in T. These other frequencies are called the *harmonics* of the signal.

Putting these properties together we conclude that the Fourier transform $F(u)$ of a periodic function $f(x)$ yields a discrete function with non-zero function values at the fundamental frequency f_T and some harmonics $k \cdot f_T$, $k \in \mathbb{N}$.

The application of the inverse Fourier transform equals the Fourier transform up to a complex conjugation. Therefore, the inverse Fourier transform of a periodic function is also discrete, and hence the Fourier transform of a discrete function is periodic.

Combining the properties of both transformations we see that the Fourier transform of discrete and periodic functions will again be discrete and periodic. For this reason, we will periodically continue all finite discrete functions implicitly from now on.

Using periodic discrete functions, we can discretize both the Fourier transform and the inverse Fourier transform (cf. Eq. 15.1 and 15.2) in the following way:

$$F_k = \frac{1}{n} \cdot \sum_{i=0}^{n-1} f_i \, \omega_n^{-ik} \tag{15.6}$$

$$f_i = \frac{1}{n} \cdot \sum_{k=0}^{n-1} F_k \, \omega_n^{ik}, \tag{15.7}$$

where $\omega_n := e^{i\frac{2\pi}{n}}$ is the nth root of unity, i.e. $\omega_n^n = 1$. If we define the discrete convolution operator (cf. Eq. 15.3)

$$(f * g)_i := \sum_j f_j g_{i-j}$$

we can derive the discrete convolution theorem (cf. Eq. 15.4):

$$\begin{array}{rcl} (f * g)_i & \circ\!\!-\!\!\bullet & F_i \cdot G_i \\ (F * G)_i & \bullet\!\!-\!\!\circ & f_i \cdot g_i. \end{array}$$

15.2 Finite Impulse Response Filters

Finite Impulse Response filters (FIR filters) are, in general, transfer functions having finite support. This restriction obviously reduces the complexity of applying them, since most of the terms in the discrete convolution $\sum_i f_i g_{k-i}$ vanish.

For reasons of simplicity, transfer functions are conveniently specified using a *filter mask* of weights, which are transformed into a transfer function by normalizing their weights. If a mask is

$$m = [\alpha_0 \alpha_1 ... \alpha_{n-1}],$$

the corresponding transfer function would be

$$g = \frac{1}{\sum \alpha_i} \cdot [\alpha_0, \alpha_1, ..., \alpha_{n-1}].$$

As stated in the last section already, filters can be designed by actually specifying their Fourier transformed. That is, we design a function G in the frequency domain, such that multiplying a given frequency spectrum F by it has the desired effect on F, like e.g. attenuating high frequencies. The corresponding transfer function g in the spatial domain can then be obtained by calculating the inverse discrete Fourier transform of G.

15.2.1 Averaging Filters

Averaging filters are low-pass filters removing high frequencies from a signal. In case of images, regions with a constant or very similar color would not change, whereas in regions of rapid color changes these differences would be attenuated.

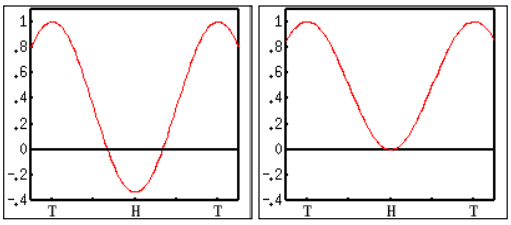
The *uniform averaging filter* replaces a value of the signal by the average of its one-neighborhood. In the one-dimensional case this corresponds to the operation

$$f_i \longleftarrow \frac{1}{3}(f_{i-1} + f_i + f_{i+1}),$$

or written as a filter mask

$$m_{avg} = [1 \; 1 \; 1].$$

Let us look at what the uniform filter does in the frequency domain. Its Fourier transformed is $F(u) = \frac{1}{3}(1 + 2\cos u)$, as depicted in the left figure. It has indeed the form of a low-pass filter, since high frequencies are attenuated, but it unfortunately drops below zero. Thus, high frequencies will not be reduced to zero completely.



The following *weighted averaging filter* does not have this flaw:

$$m_{wavg} = [1\ 2\ 1]\,,$$

resulting in the Fourier transformed $F(u) = \frac{1}{4}(2 + 2\cos u) \geq 0$. The right figure shows this function. Note that the function does not drop beneath zero anymore, and hence high frequencies are completely eliminated from the signal.

15.2.2 Differencing Filters

In contrast to averaging filters, *differencing filters* are high-pass filters, removing low frequencies from a signal. By applying such filters to images, homogeneous regions are attenuated, whereas strong changes are amplified. Therefore, these filters can be understood as edge detectors, since edges in the image are the high-frequent regions that are intensified.

The first derivative of the function can be approximated by the central difference

$$f_i \longleftarrow \frac{1}{2} \cdot (f_{i+1} - f_{i-1})$$

and therefore can be represented by the mask

$$m_{diff} = [-1\ 0\ 1]\,.$$

This differencing filter therefore approximates the first derivative, which is high at edges and takes small values in homogeneous regions. Hence it represents the high-pass edge detection filter we were looking for.

15.2.3 Combining Filters

The effects of two filters can easily be combined by adding the corresponding two masks. Consider e.g. the filter $[-1, 2, -1]$. It can be written as the following sum of filters:

$$[-1\ 2\ -1] = [0\ 4\ 0] - [1\ 2\ 1]\,.$$

The first term is the identity, and from this a low-pass filter is subtracted. Subtracting the low frequencies from the identity in fact yields a high-pass filter.

Using the very same technique, one can create a custom bandpass filter, i.e. a filter preserving only a certain range of frequencies by subtracting a high-pass and a low-pass filter from identity.

15.3 Two-Dimensional Filters

In image processing we are facing two dimensional functions, because 2D pixel positions have to be mapped to their corresponding colors. Hence, we must generalize the 1D filters we derived in the previous section to the two-dimensional case.

15.3.1 2D Discrete Fourier Transform

In order to handle 2D filters we first have to extend the discrete Fourier transform to the 2D case. Fortunately, this can be done straight-forward (cf. Eq. 15.6):

$$\begin{aligned} F_{k,l} &= \frac{1}{m}\sum_{i=0}^{m-1}\omega_m^{-ik}\frac{1}{n}\sum_{j=0}^{n-1}\omega_n^{-jl}f_{i,j} \\ &= \frac{1}{nm}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}\omega_m^{-ik}\omega_n^{-jl}f_{i,j} \end{aligned}$$

Using the 2D Fourier transform, we can now transform the pixels of an image to their frequency spectrum. Note that mn pixels lead to mn distinct Fourier coefficients $F_{k,l}$; one normally depicts them in a gray-scale image, choosing the brightness according to the magnitude of the Fourier coefficients (see figure).

15.3.2 2D Filter Design

In the two-dimensional case the filter masks are matrices instead of vectors, i.e.

$$M = \begin{bmatrix} \alpha_{0,0} & \dots & \alpha_{0,n-1} \\ \vdots & & \vdots \\ \alpha_{n-1,0} & \dots & \alpha_{n-1,n-1} \end{bmatrix}.$$

Note that we can separately look at the two dimensions of an image. That is, we can first perform a 1D-filter with mask m_1 vertically, followed by the application of another 1D-filter with mask m_2 horizontally. The 2D-mask representing both filters is then their product:

$$M = m_1^T \cdot m_2$$

15.3.3 Averaging Filters

Based on the 1D masks presented in Sec. 15.2.1 we get the corresponding 2D masks by multiplying the 1D masks by themselves.

For the two-dimensional uniform averaging filter we obtain the mask

$$M_{avg} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \cdot [1\ 1\ 1] = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix},$$

corresponding to the transfer function

$$f_{i,j} \longleftarrow \frac{1}{9}(f_{i-1,j-1} + f_{i-1,j} + f_{i-1,j+1} + f_{i,j-1} + f_{i,j} + f_{i,j+1} + f_{i+1,j-1} + f_{i+1,j} + f_{i+1,j+1}).$$

Analogously, we obtain the 2D weighted averaging filter (cf. Fig. 15.2c):

$$M_{wavg} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot [1\ 2\ 1] = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}.$$

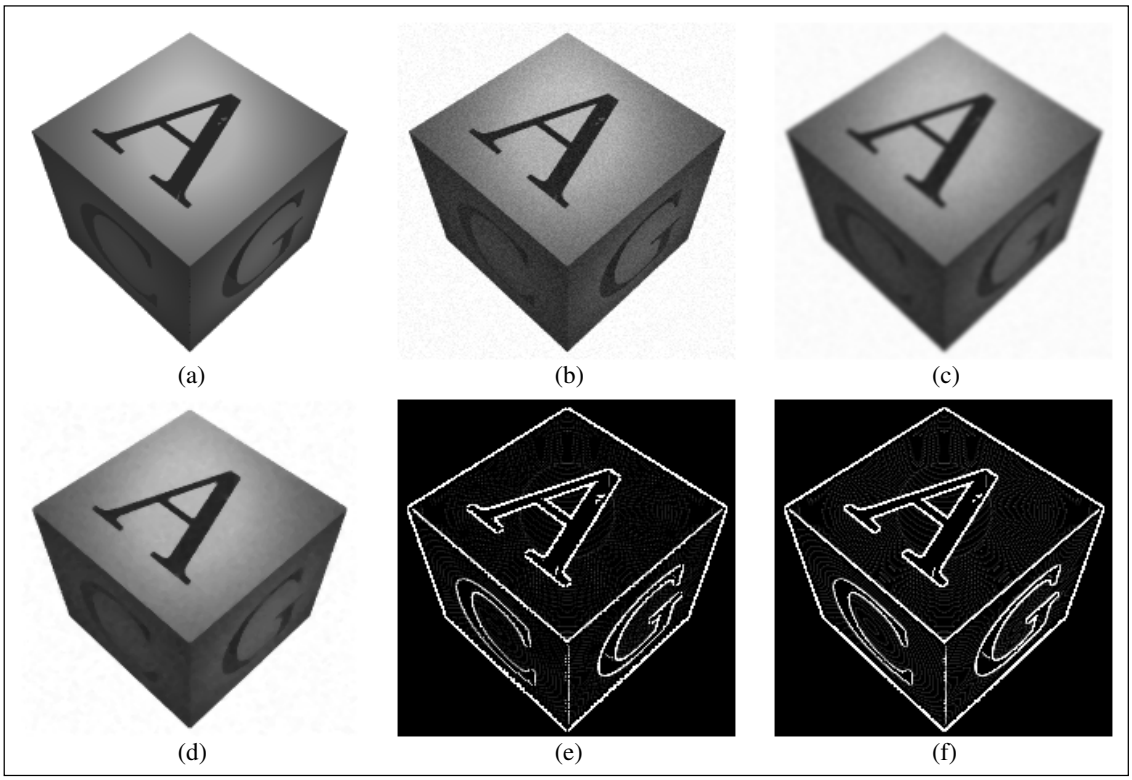


Figure 15.2: Image Filters: (a) Original image, (b) noise added to a, (c) weighted averaging of b, (d) median filtering of b, (e) Sobel of a, (f) Laplace of a.

15.3.4 Differencing Filters

Combining the 1D differencing filter of Sec. 15.2.2 in one direction with an uniform averaging filter in the other direction (causing a higher stability) results in 2D high-pass (i.e. edge detection) filters. These so-called *Prewitt filters* represent partial derivatives w.r.t. the u and v direction:

$$\begin{aligned} M_{Prewitt,u} &= \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \cdot [-1\ 0\ 1] = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \\ M_{Prewitt,v} &= \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \cdot [1\ 1\ 1] = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}. \end{aligned}$$

Again, using the weighted averaging filter instead of the uniform averaging filter improves quality, and the resulting filters are called *Sobel filters* (cf. Fig. 15.2e):

$$\begin{aligned} M_{Sobel,u} &= \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \cdot [-1\ 0\ 1] = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\ M_{Sobel,v} &= \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \cdot [1\ 2\ 1] = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \end{aligned}$$

Both differencing filters discussed so far used the first derivative. However, we could also use the

second derivative in both directions, yielding the *Laplace filter* (cf. Fig. 15.2f)

$$M_{Laplace} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

15.4 Alias Errors

Images can be considered to sample a 2D function on a regular pixel grid. Since any regular sampling is likely to cause aliasing artifacts, we also have to take this topic into account for image processing. We already discussed this matter in the context of texture anti-aliasing (cf. Sec. 4.5.5) and ray tracing (cf. Sec. 14.3). Here we will look at the reason for the alias errors and at techniques to avoid them in general. The Fourier transform provides use with the mathematical tools to better understand the alias problem.

Sampling a continuous function f corresponds to multiplying it by a discrete function s consisting of shifted *impulses* $\delta(x)$ with

$$\delta(x) = \begin{cases} 1, & \text{if } x = 0 \\ 0, & \text{otherwise} \end{cases}.$$

Assuming a constant sampling rate Δx, we get (cf. Fig. 15.3a)

$$g(x) = \sum_{i=-\infty}^{\infty} \delta(x - i\Delta x).$$

We can furthermore assume that the frequency spectrum of f is bounded by a frequency ω_{max}, i.e. the support of its Fourier transformed F is $[-\omega_{max}, \omega_{max}]$.

Note that this is a realistic assumption; and even if this is not the case, we can usually cut off very high frequencies without changing the signal f significantly. It can be shown that the Fourier transformed of s is again a series of impulses, but this time with a spacing of $1/\Delta x$:

$$s(x) \circ\!\!-\!\!\bullet\, S(u) = \sum_{i=-\infty}^{\infty} \delta\left(u - \frac{i}{\Delta x}\right).$$

Sampling the function f leads to the function $f \cdot s$. Due to Eq. 15.4, multiplication in spatial domain corresponds to convolution in frequency domain. Hence, the Fourier transformed of $f \cdot s$ is a function in which $F(u)$ is replicated at each position of an impulse in $S(u)$.

If the support of F is larger than the distance between two consecutive impulses in S, these copies of F overlap. In regions where this happens it is impossible to reconstruct the original signal f, since its frequency spectrum changed. This is the reason for alias errors when using discrete sampling (cf. Fig. 15.3a).

The *Nyquist Condition* states exactly this correspondence between sampling rate and signal reconstruction: a correct reconstruction can only be ensured, if

$$\Delta x \leq \frac{1}{2\omega_{max}}.$$

Since the sampling rate is fixed for an image, we need to adjust its maximum frequency ω_{max}. This can be done by applying a low-pass filter, which cuts off frequencies higher than ω_{max} before sampling the image (cf. Fig. 15.3b).

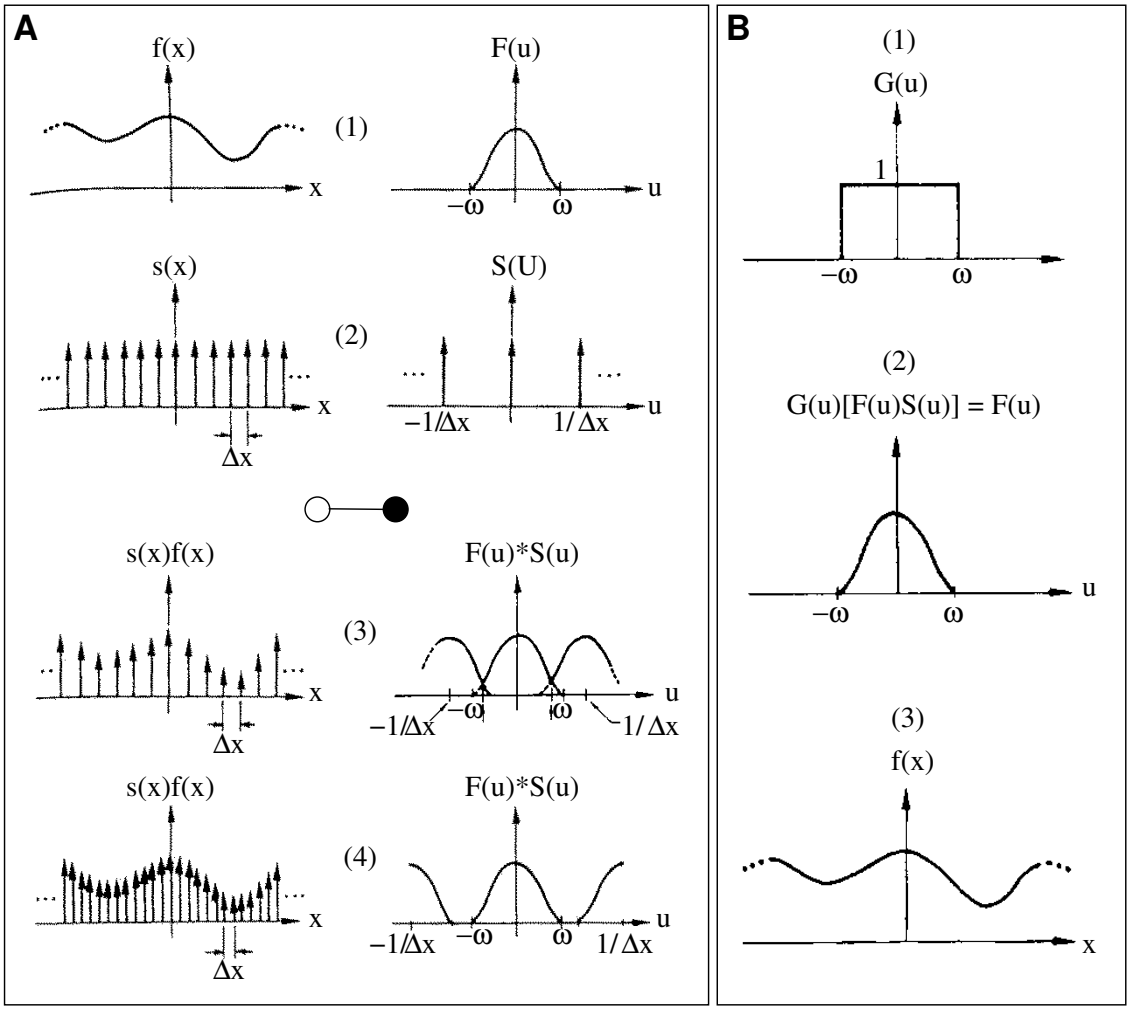


Figure 15.3: Alias Errors due to Sampling: (a) Sampling: 1 shows the function to be sampled, 2 the transfer function, 3 and 4 show the convolution result with under-sampling in contrast to a sufficient sampling rate. (b) Reconstruction: 1 shows the low-pass filter, 2 the convolution result after applying the filter, and 3 the reconstructed signal.

We have seen this already when we discussed texture anti-aliasing in Sec. 4.5.5. Note that integrating over the area of the new samples is in effect a low-pass filtering procedure.

When rendering an image, we can also use super-sampling to avoid alias effects: we simply sample the scene on a sub-pixel basis, apply a low-pass filter to the image and finally subsample it to the resolution of the pixel grid (see also chapter 14.3).

15.5 Non-Linear Filters

In this section we will discuss non-linear filters, which, in contrast to linear filters, cannot be expressed in terms of linear convolutions. Although they are in general computational more expensive, they provide more powerful image processing operations.

15.5.1 Perona-Malik Smoothing

The *Perona-Malik filter* is based on the Laplace filter (cf. Sec. 15.3.4). Recall that the Laplace filter mask is

$$M_{Laplace} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Also it is obvious that the following mask represents the identity:

$$M_{identity} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

The *Perona-Malik* filter combines these two masks using an adaptive factor $\lambda \in [0, 1]$:

$$M_{PM}(\lambda) = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \lambda \frac{1}{8} \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \frac{1}{8} \begin{bmatrix} \lambda & \lambda & \lambda \\ \lambda & 8 \cdot (1-\lambda) & \lambda \\ \lambda & \lambda & \lambda \end{bmatrix}.$$

Using the filter $M_{PM}(0)$ yields the identity, whereas using $M_{PM}(1)$ is a very effective low-pass-filter, since it replaces the colors of the pixels by the average of their neighbors. The idea is to locally adapt λ according to the neighborhood: in regions of high contrast (important edges) we choose $\lambda \approx 0$, and if there is low contrast and noise instead, we choose $\lambda \approx 1$. Hence, this filter removes noise in homogeneous regions without blurring sharp edges in regions of high contrast.

Note that this filter is not linear, although for each specific value of λ it actually is. However, λ is adjusted according to the local neighborhood of the pixels and therefore cannot be expressed linearly.

15.5.2 Median Filter

The *Median* filter is also some kind of low-pass filter. It replaces a pixel's color by the median of the pixel colors in its neighborhood. Recall that the *median* of a set is an element from that set, such that half of the other elements are smaller, and half of them are greater. For example, $Median(\{1, 4, 6, 9, 11\}) = 6$.

Now consider the k-neighborhood of a given pixel position:

$$N_k(i, j) = \{(i', j') \,:\, |i - i'| \leq k \wedge |j - j'| \leq k\}.$$

Then the median filter assigns to each pixel (i, j) the color

$$color(i, j) \longleftarrow Median(color(N_k(i, j))).$$

Applying this filter to an image results in images similar to those obtained using an averaging filter. However, there are three important differences. First, the new color of a pixel already exists in the neighborhood. Second, the median filter is less sensitive to outliers than an averaging filter. And finally, it does not blur sharp color edges, but preserves them quite well (cf. Fig. 15.2d).

Therefore, this filter is ideally suited to remove noise and outliers from an image. Note that although it is more stable than the averaging filter, it has the disadvantage of being non-linear and therefore is more expensive to compute.

15.5.3 Histogram Equalization

In contrast to the Perona-Malik and median filter, *Histogram* filters are not based on the neighborhood of a pixel, but on its color only.

Recall that a histogram counts the number of occurrences of a certain value. In this case we count the number of occurrences of a certain color, i.e. in an image containing K colors, we compute the following values:

$$n_k := |\{(i, j) \,:\, color(i, j) = k\}|, \quad k = 0, .., K - 1.$$

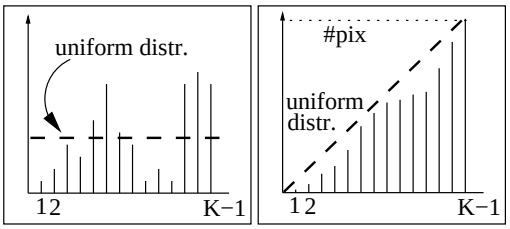
These values now represent the histogram, see left figure below. Note that we do not use spatial information here: we only know how often a color appears in an image, but not at which parts of the image it occurs.

Histogram filters assign to each pixel a new color based on its old color (not its position in the image) and on the histogram. The transfer function gets

$$T : \{0, ..., K - 1\} \longrightarrow \{a, ..., b\}$$

where $\{a, ..., b\}$ represents a range of colors onto which the K original colors are mapped. In order to keep the color tones, histogram filters normally operate on intensities of colors. This means, the histogram is computed based on the brightness level of the pixels, and then the pixel colors are brightened or darkened due to the filter without changing the ratio of the color components.

A *Histogram Equalization* filter tries to evenly distribute the colors in the image, i.e. it tries to achieve a constant histogram function (see left figure). Then, the image reveals most information, since the most possible extent of differentiation between pixels has been reached.



Unfortunately, based on the histogram $\{n_i\}$ this is hard to achieve. Therefore, we compute the accumulative histogram (see right figure above)

$$\tilde{n}_k := \sum_{i=0}^{k-1} n_i, \quad k = 0, ..., K - 1.$$

Note that $\tilde{n}_{K-1} = \#pixels$, since this value corresponds to the number of pixels having any color. The ideal distribution of colors in the accumulative histogram chart would result in a line of slope one. We now can map the pixel colors easily to a position near to the line:

$$T(i) = \left\lfloor K \cdot \frac{\tilde{n}_i}{\#pixels} \right\rfloor.$$

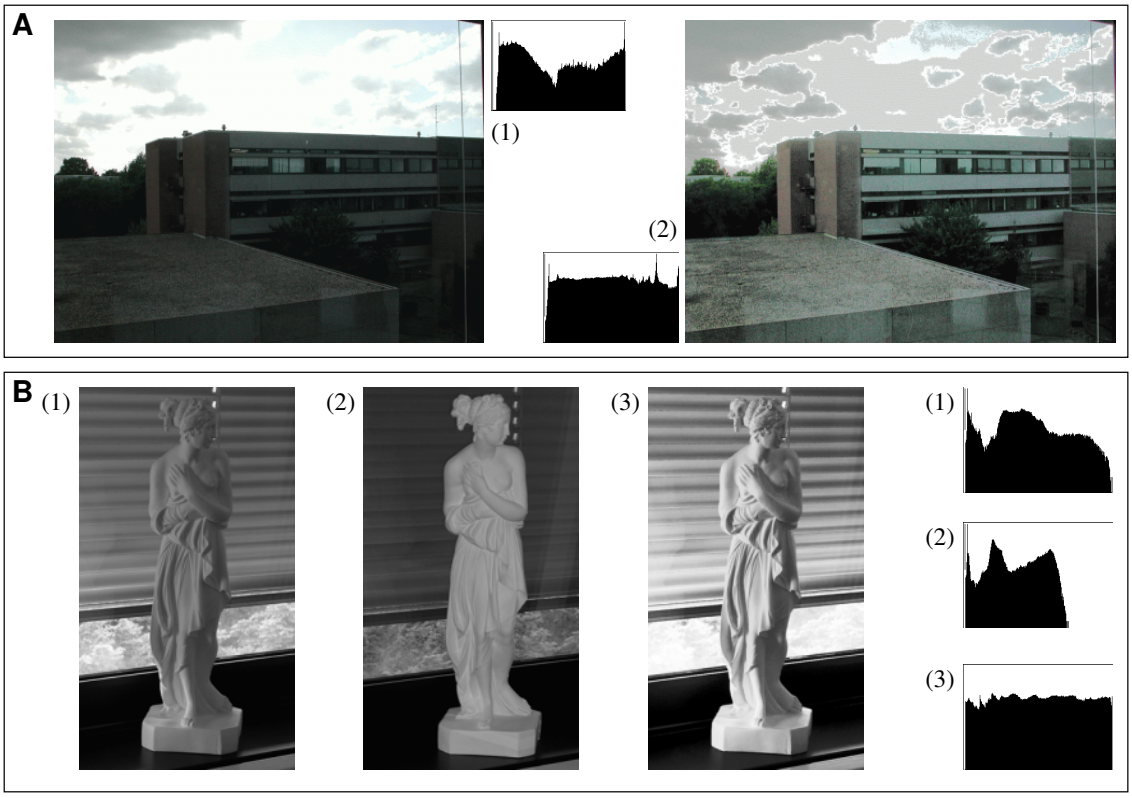


Figure 15.4: Histogram Equalization: (a) picture of the I8 department, without (1) and with (2) histogram equalization, (b) picture of a statue taken without flash (1), with flash (2), and the histogram-equalized version (3).

Applying this filter to an image will distribute the colors (almost) evenly, and will therefore achieve the maximum possible contrast in the picture. Areas in which almost constant colors were before will spread over a larger color range, improving the visual contrast of the image. Note however, that due to this process images might appear less realistic or natural (cf. Fig. 15.4).

15.5.4 Morphological Operators

Morphological operators classify pixels into an active and a passive set and add pixels to or remove them from this set according to a given stencil mask (called the *structure element*). First, all pixels of the image

$$I = \{(i,j) \,:\, 0 \le i < m, 0 \le i < n\}$$

are grouped into two disjunct subsets. For example, we could do this according to a specific set of colors C, splitting the image into fore- and background:

$$\begin{aligned} A &= \{(i,j) \,:\, color(i,j) \in C\} \\ A^\star &= \{(i,j) \,:\, color(i,j) \notin C\} = I \setminus A. \end{aligned}$$

To apply a morphological operator, we also have to define its structure element M. This is a set defining pixels around some center pixel $(0,0)$, specified by their offsets,

$$M = \{(\Delta i_1, \Delta j_1), \ldots, (\Delta i_n, \Delta j_n)\}\,.$$

For example, the three pixels on the upper right corner would be described by the structure element $\{(1,0),(1,1),(0,1)\}$ (cf. Fig. 15.5a).

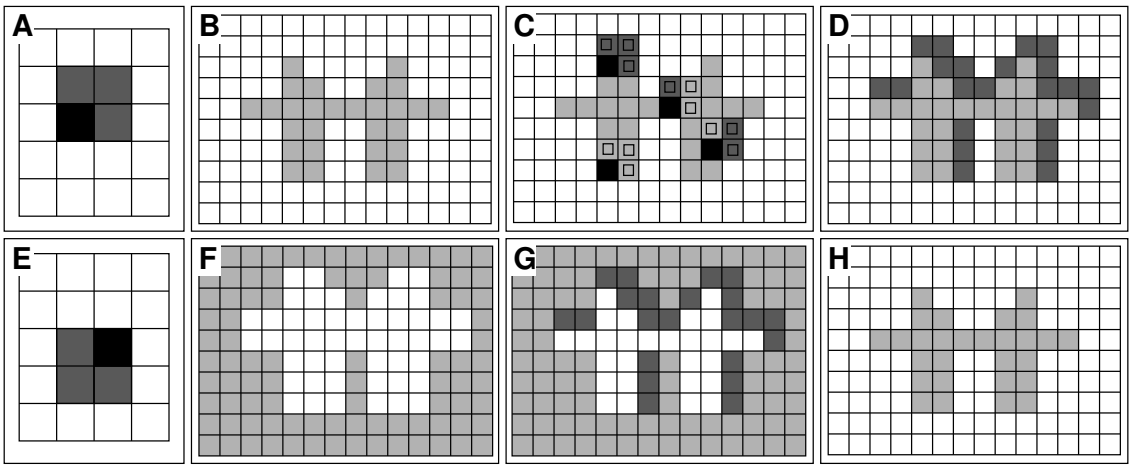


Figure 15.5: Dilation and Erosion: (a) structure element, (b) pixel set, (c) adding pixels according to structure element, (d) dilation result, (e) inverse structure element, (f) inversion of pixel set, (g) inverse dilation on inversion (h) erosion result.

The first morphological operation is the *dilation*. It adds all pixels in the neighborhood of pixels of A which are referenced by the structure element M (cf. Fig. 15.5b-d):

$$A \oplus M := \{(i + \Delta i, j + \Delta j) \,:\, (i,j) \in A, (\Delta i, \Delta j) \in M\}\,.$$

The complementary operation is the *erosion*, which deletes pixels from A according to the structure element M. This can be described by applying the dilation to the complement of A based on the inverse of the structure element M and complementing the result again (cf. Fig. 15.5e-h):

$$A \ominus M := I \setminus \{(i - \Delta i, j - \Delta j) \,:\, (i,j) \in A^\star, (\Delta i, \Delta j) \in M\}\,.$$

Based on these basic operators we can now define two more useful operations, called *opening* and *closing*:

$$\begin{aligned} \text{Closing} &: (A \oplus M) \ominus M \\ \text{Opening} &: (A \ominus M) \oplus M \end{aligned}$$

The closing operator closes small gaps in A, whereas the opening operator cuts off thin needles of the object, as illustrated in Fig. 15.6. This also shows that dilation and erosion are in general not inverse operations. If the dilation completely fills parts inside or near the object, erosion will not be able to open these parts again, and, conversely, if erosion cuts off parts of the objects completely, dilation will not be able to reconstruct them.

Opening and closing operations can be used to smoothen contours of objects, by filling small gaps and removing small peaks in its boundary.

15.6 Geometric Transformations

There is another class of important transformations on objects, namely *geometric transformations*. The filtering operations discussed so far analyzed the frequency structure of images and changed the colors of the pixels. In contrast to that, geometric transformations change the position of the pixels, e.g. by rotations or translations.

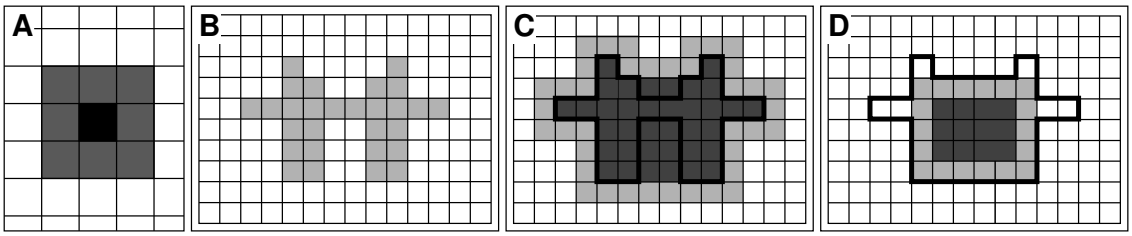


Figure 15.6: : Opening and Closing: (a) structure element, (b) pixel set, (c) closing of b, (d) opening of c. In (c) and (d), the initial pixel set is marked with a line, the light color refers to the dilation step, and the dark color to the erosion step.

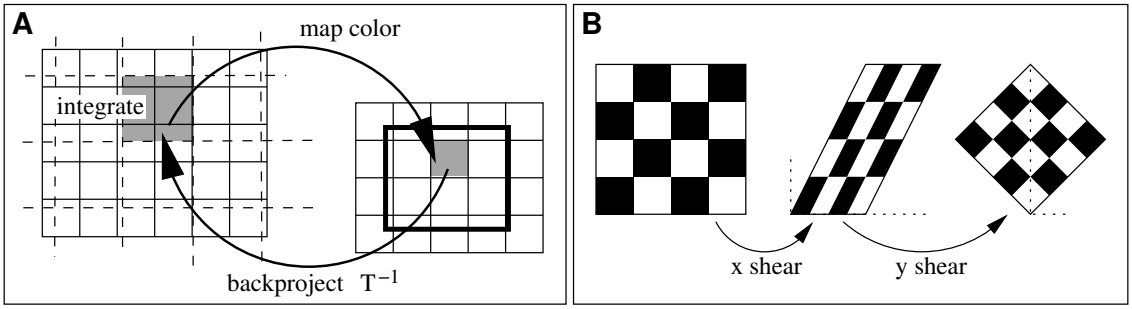


Figure 15.7: Geometric Transformations: (a) computing the transformed image, (b) multi-pass transformations.

A very basic approach would be to map each pixel of the source image to the target image using a given transformation T, and to fill the target pixel with the corresponding color.

However, this approach does not work in general: we get problems whenever one pixel in the source image is mapped to several pixels in the target image, or vice versa. As a consequence, pixels might not get filled at all, resulting in gaps, or they might be filled multiple times.

Therefore, we use a back-projection technique instead. For each pixel in the target image, we inversely transform its position, and integrate over the region of its pre-image in the source image. The color obtained in this way is then assigned to the target pixel (cf. Fig. 15.7a).

Integration in this discrete context means to calculate the weighted average of the source pixels which are at least partially inside the pre-image of the target pixel. The weights are obviously chosen according to the area a source pixel occupies in the pre-image.

The problem with this approach is caused by the back-projection; we need to calculate which area the pre-image of a pixel is being mapped to, and we need to find the pixels in the source image which are inside that pixel, together with the amount of the pre-image they occupy.

15.6.1 Multi-Pass Transformations

Fortunately, we can accelerate the computation of the target pixel colors by decomposing the transformations analogously to what we did in the Shear-Warp-Factorization (cf. Sec. 10.2.3).

Note that each of the basic transformation operations can be expressed in terms of vertical and horizontal shears, potentially combined with scaling (cf. Fig. 15.7b).

Horizontal and vertical shears (scaled or not) are row and column preserving, respectively. Therefore, we only need to compute an offset and a scaling factor per row/column, simplifying the 2D transform to a set of 1D transforms.

It is very important to see why this method is so advantageous: since the shearings are row or column preserving we can exploit scanline coherence, resulting in very efficient implementations.

Chapter 16

Color Coding

Recall from chapter 2.2.2 that there are several color models we can use for representing the color of pixels in an image, like e.g. RGB, HSV and CIE. In this chapter we will focus on the color model most commonly used for images, i.e. on the RGB model. However, the methods described here can be used for other color models as well.

In order to be able to represent almost every visible color (so-called *true color*) we spend 8 bit precision for each of the three color channels, leading to a total of 16 million distinguishable colors. Unfortunately, storing an image with this color resolution requires 3 bytes per pixel, leading to a high memory consumption.

In this chapter we will discuss color reduction techniques that try to keep the visual quality of images while using fewer colors. The next chapter will then focus on image compression in general.

16.1 Color Reduction

There are generally two ways to decrease the number of colors in an image:

- *Quantization* refers to changing the resolution of the color channels, thereby obviously reducing the size required to store a pixel color. Note that even true color is already a quantization into 16 million colors. However, this number is high enough to give the impression of "real" colors.
- *Color Tables*: Since most pictures use a relatively small number of colors only (compared to the 16 million available colors), another way to reduce memory consumption is to create a color table, and then reference colors by their indices into that table. An important issue in this respect is which colors to choose for the table.

In the following sections we will describe both reduction techniques. Note however that quantization will in most cases lead to a reduction of image quality, whereas color tables might not if the image contains sufficiently few colors, which can all be represented by the color table.

16.1.1 Quantization

Quantization evenly samples the color channels into a discrete set of values. That is, the color function changes from

$$F : \{0, ..., w-1\} \times \{0, ..., h-1\} \longrightarrow \mathbb{R}^3$$

to

$$F : \{0, ..., w-1\} \times \{0, ..., h-1\} \longrightarrow \{0, ..., r-1\} \times \{0, ..., g-1\} \times \{0, ..., b-1\}$$

True color, which we assumed so far for images, is actually also a quantization, but fine enough to evoke the impression of synthesizing all colors. More precisely, true color usually refers to $r = g = b = 256$, allowing for 16 million colors. It obviously requires three bytes per pixel.

A commonly used quantization is *R3G3B2*. The numbers refer to the resolution of the color channels in bits. This means, the red and green channel distinguish eight tones each, whereas the blue channel only distinguishes four colors. Note that each color can now be stored using eight bits (one byte), a very common size. In this quantization we choose the blue channel to have the lower precision, because the eye reacts much less sensitive on blue than on red or green, due to the distribution of color receptors in the eye (cf. Sec. 2.2.1).

Note that now only 256 different colors can be displayed, with the colors being evenly distributed in the color cube. Hence, image regions with small color deviations may be mapped to one color only, even if the whole image only contains less than 256 different colors. However, one can easily change a pixel's color to one of the available colors; in case of a color table this would require to potentially enter or replace a color in the table.

16.1.2 Color Tables

Color Tables are another very common way to reduce the size of an image. Instead of specifying a color for each pixel, we store a table mapping indices to colors, and then specify an index per pixel referencing a color in the table:

$$\begin{array}{rccc} c: & \{0, ..., k-1\} & \longrightarrow & \{0, ..., 255\}^3 \\ & i & \longmapsto & (r_i, g_i, b_i) \end{array}$$

$$\begin{array}{rccc} f: & \{0, ..., w-1\} \times \{0, ..., h-1\} & \longrightarrow & \{0, ..., k-1\} \\ & (u, v) & \longmapsto & i \end{array}$$

The number of colors k is commonly chosen to be 256, since then every color can be referenced by just one byte, and in most cases a reduction to 256 well chosen colors does not reduce the image quality too much.

Obviously, the most important issue for using color tables is the generation of an optimal color table for a given image. Optimal, in this context, refers to a minimum deviation of the color-reduced image from the the original one.

Of course, finding an optimal solution for the problem is very hard and algorithmically not feasible. However, there is a heuristic leading to very good results: the *median-cut algorithm*.

16.1.3 Median-Cut Algorithm

The median-cut algorithm is a recursive process, consisting of the following steps:

1. Histogram: A histogram of image colors is computed measuring which colors are used how frequently in the image.
2. Bounding-box: A bounding box in RGB space is placed around the resulting histogram entries.
3. Subdivision: This color bounding box is recursively subdivided until the number of boxes equals the number of entries in the color table.

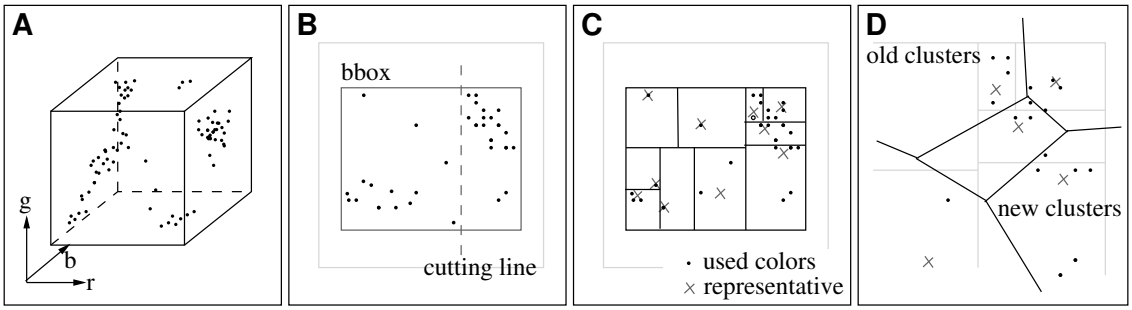


Figure 16.1: Median-Cut: (a) histogram, (b) bounding box and median cut, (c) computation of representatives, (d) iterative clustering.

4. Representatives: In a last step, the algorithm computes a representative color for each box, and assigns this representative to all colors contained in this box.

5. Iterative Clustering: The representatives might be iteratively optimized, if desired.

In the remainder of this section we will discuss these steps in more detail. Note that the illustrative figures partially show the 2D case for simplicity.

Creating a Histogram

The first step is to create a histogram of the colors in the image. Since true color uses eight bit per color channel, we create a three-dimensional array of 256^3 entries, and associate with each entry the number of pixels in the image having that color (cf. Fig. 16.1a).

Afterwards, we are of course interested in only those entries having a value greater than zero, since these represent the colors used in the image. We now compute a bounding box around these non-zero entries in the 3D histogram that will be split into smaller boxes recursively.

Median Cutting

We split the box along its longest side/edge into two children by inserting a cutting plane orthogonal to this side (cf. Fig. 16.1b). Cutting through the longest edge is preferable, since along this edge the highest color deviation is encountered.

The splitting position is chosen such that the colors on either side of the cutting plane are referenced by equally many pixels in the image. Hence, there are equally many non-zero histogram entries on both sides (counting multiplicities).

The algorithm now proceeds by splitting the sub boxes recursively at their longest edges. Each split increases the number of cells/colors by one. We proceed until the number of leaves in the tree equals the number of colors to be used in the color table.

Computing the Representatives

For each resulting box we now have to compute a representative color. This could e.g. be the weighted average of the colors found in the box or the weights chosen according to the multiplicity of the colors in the image (cf. Fig. 16.1c).

The color table is now filled with these representatives, and all colors in the image are replaced by the representative color of their corresponding box.

Iterative Clustering

The representatives might not be optimal at this point. Note that due to the way the representatives were computed, it might happen that a representative of another box is closer to a color than the representative of its own box.

Therefore, we apply an iterative improvement. After computing the representatives, we assign to each color the closest representative (as opposed to the one of its box). This leads to new, potentially different clusters of colors (cf. Fig. 16.1d).

We therefore proceed by computing representatives for the new clusters by computing their weighted averages. Again, assigning all colors in the cluster to the new representative might not be optimal, since other representatives could be closer. Thus, we can again refine the choice of representatives using the same technique.

Iteratively, we will find better and better representatives. We can stop this process when the new representatives do not deviate much from the old ones.

16.2 Dithering

In some situations the output device might not be able to display all colors the image contains, e.g. in the case of a cell phone's 256 color display. Another prominent example are printers: here, only one color (black) can be printed on white paper.

The techniques presented in this section are used to simulate gray-scales. Although we will only look at gray-scale images (with a 256 color scale), the methods can be extended to color images by simply applying the technique to each of the color channels.

16.2.1 Naive Approaches

In this subsection we will introduce some very simple methods to improve the visual impression of images using just a small number of colors. As we will see, the image quality obtained by these techniques is not sufficient, but these methods serve as an understanding of what has to be improved.

Down Scaling

A very simple method to assign gray values to the pixels in the color-reduced image is to make a per-pixel binary decision. That is, we map the original colors evenly to the colors of the reduced palette. In case of a reduction to black and white, we would assign white color to all gray intensities of at least 128, and black color otherwise.

The results are shown in Fig. 16.2b. Note that we lose the smoothly shaded regions in the image, because the color jumps from black to white at once.

Random Decision

One way to improve this is to use a random generator. Instead of taking a fixed decision according to the original pixel's value, we only use this value as a probability that this pixel will or will not be set. For example, a pixel with gray intensity 152 gets probability 152/256 to be filled with white color.

Although this random generator will replace the hard edges between black and white by noisy blending, we would need a very good random number generator to achieve acceptable results.

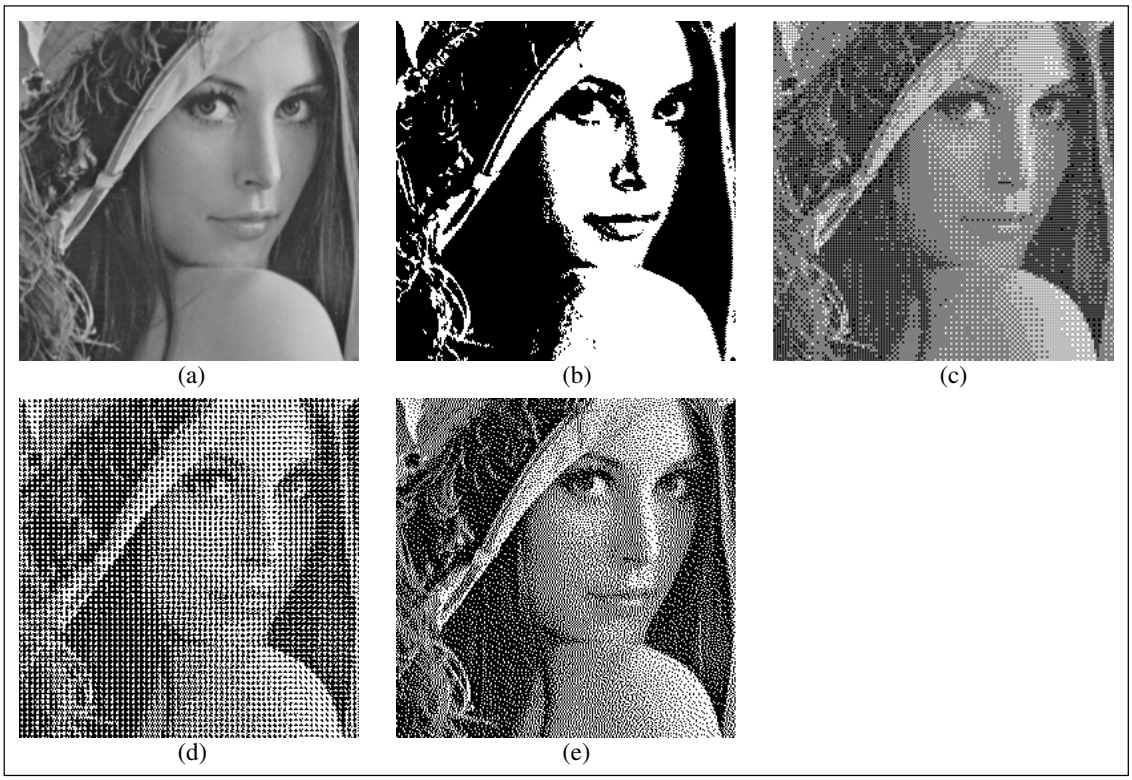


Figure 16.2: Dithering: (a) original image, (b) black white version, (c) dispersed dots dithering, (d) clustered dots dithering, (e) Floyd-Steinberg error diffusion.

Usually black pixels will have the tendency to cluster together in an irregular pattern, resulting in low image quality.

Filling patterns

Filling patterns are a way to use a regular pattern, but still allow for a number of gray tones. We could, e.g., use 2×2 filling patterns (cf. Fig. 16.3a) in order to simulate five different gray levels. Generally, a $n \times n$ pixel pattern can be used to simulate $n^2 + 1$ different gray intensities.

Using the 2×2 pattern we combine image pixels to groups of 2×2 pixels each. Then, for each of these groups, we compute their average gray level and fill them using a corresponding pattern. Unfortunately, this technique reduces the resolution, since in effect a number of pixels is given the same gray intensity.

We will therefore try to find a way to combine all three naive approaches, i.e. make a decision based on a given pixel, but still take into consideration its local neighborhood.

16.2.2 Dither Matrices

One way to choose a color for each pixel — while considering its neighborhood as well — is to use dither matrices. This technique tries to hide quantization artifacts by introducing noise.

Recall that in the down scaling approach the color is chosen according to the following formula (assuming that k^2 colors can be displayed):

$$(x, y, C) \longmapsto (C \text{ div } k^2).$$

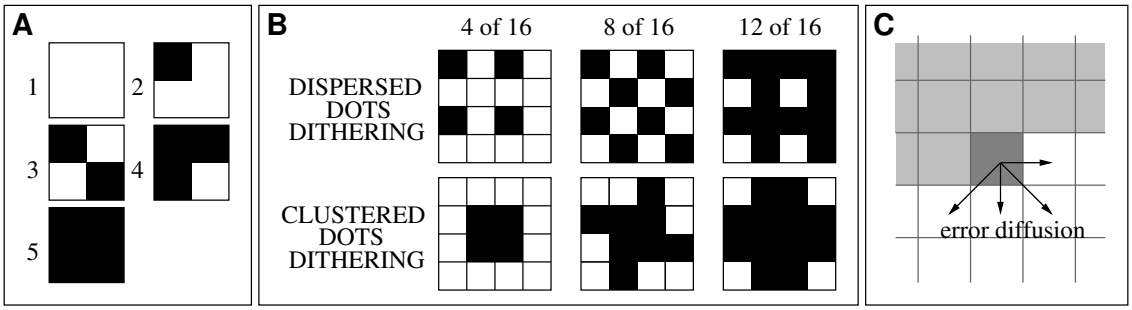


Figure 16.3: Dithering: (a) filling patterns, (b) dither schemes in homogeneous regions, (c) Floyd-Steinberg.

Note that no neighborhood information is used to obtain that value.

Dither matrices extend this scheme by adding a pattern adjusting the down scaled version according to the error made due to rounding.

A dither matrix for a reduction by a factor of k^2 is a $k \times k$ matrix with entries $0, ..., k^2 - 1$. For example, to reduce the number of colors by a factor of 16, the matrix might look like this:

$$D = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}.$$

Using this dither matrix, we now map a pixel to its corresponding matrix entry using modulo-arithmetic. Due to rounding errors in the down-scaling process the color C at pixel (x, y) falls between

$$(C \text{ div } k^2) \text{ and } (C \text{ div } k^2) + 1.$$

We choose between these two based on the remainder $C \bmod k^2$ and the pixel position (x, y):

$$(x, y, C) \longmapsto (C \text{ div } k^2) + \begin{cases} 0, & \text{if } C \bmod k^2 < D_{(x \bmod k, y \bmod k)} \\ 1, & \text{otherwise} \end{cases}$$

Note that the color decision is mainly based on one pixel only, however it is slightly corrected according to a neighborhood pattern. We will now describe two special dithering patterns. Of course, other effects can be achieved by changing the dither pattern.

Dispersed Dots Dithering

The matrix D we have seen above corresponds to a dithering scheme called *dispersed dots*:

$$D_{dispersed,4} = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}.$$

Note that in regions of constant color, the dither matrix causes pixels to be filled by a regular pattern (cf. Fig. 16.3b). If the original color is constant, then the pattern for any 4×4-region of pixels will be the same. Note that using this scheme we retain the original resolution, since the spatial influence of a pixel on its final color is dominant. Dispersed dot matrices can be computed for any size $2^n \times 2^n$ using a recursive pattern according to the following 2×2 matrix:

$$D_{dispersed,2} = \begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}$$

Clustered Dots Dithering

In contrast to dispersed dots dithering, the aim of *clustered dots* is not to achieve the highest possible resolution, but to achieve the highest possible contrast instead. The corresponding matrix looks like this:

$$D_{clustered,4} = \begin{pmatrix} 13 & 10 & 6 & 14 \\ 5 & 0 & 3 & 11 \\ 9 & 2 & 1 & 7 \\ 12 & 4 & 8 & 15 \end{pmatrix}.$$

Note that this pattern creates a larger and larger dot in the middle of the block (cf. Fig. 16.3b). As a consequence the resolution of the image looks like being decreased by a factor of four. This method is used for newspapers, for example. Note that other matrix sizes can be obtained by distributing the dither values in a spiral pattern from the middle of the matrix.

Fig. 16.2 shows the two dithering techniques in comparison, together with the original image, the number of gray tones reduced by a factor of 16.

16.2.3 Floyd-Steinberg Error Diffusion

In the dithering approach we try to reduce the error introduced by color reduction by having the color decision depend on the position of the pixel in the dither matrix. For large homogeneous regions, the error made locally due to rounding is in fact eliminated. However, when neighboring pixels have a different color, this technique does not work very well.

But we can in fact reduce the error to a minimum by propagating it to neighboring pixels and taking it into account when deciding about their color. This is the idea of the *Floyd-Steinberg* error diffusion algorithm.

The algorithm traverses the pixels in scanline-order. At each position, it computes the color according to the down scaling approach and calculates the error. If the original color was $C(x, y)$ and the reduced new color is $C'(x, y)$, the error is $\Delta C = C(x, y) - C'(x, y)$.

This error is now propagated to all neighboring pixels that have not been processed yet by increasing or decreasing their intensity accordingly. In the two-dimensional case it is propagated among four pixels (cf. Fig. 16.3c). The error is propagated according to weights which have been found to achieve the best results, leading to the Floyd-Steinberg algorithm:

$$\begin{aligned} C(x+1, y) &+= \frac{7}{16}\Delta C \\ C(x-1, y-1) &+= \frac{3}{16}\Delta C \\ C(x, y-1) &+= \frac{5}{16}\Delta C \\ C(x+1, y-1) &+= \frac{1}{16}\Delta C \end{aligned}$$

Note that by distributing the error to the neighbors, the overall error in the resulting image gets minimal. Also, errors are resolved locally: once the error is large enough to result in a color change of one level, the error will be respected at this point, and vanish afterwards.

Fig. 16.2 shows the results of this algorithm. Note that no raster is visible in the image, and that the full resolution is used, which makes color boundaries clearly visible. Also, color shadings are smooth.

Chapter 17

Image Compression

It is clear that images require a lot of memory depending on their size and color depth. If an image is w pixels wide, h pixels high, and its colors require k bits to be referenced, the total number of bytes required to save the image without compression would be

$$w \cdot h \cdot \frac{k}{8}.$$

Obviously, a color table requires additional space.

There are obvious ways to reduce this size. For example, one could down-scale the image (thereby reducing w and h), or use less colors (thereby reducing k). We have discussed techniques to do this in the previous chapters. But doing so degrades image quality: either we lose resolution or color details.

In this chapter we will discuss compression techniques reducing the size of the image without loss of information. In this respect, we will use some technical terms:

- *Data* refers to a sequence of symbols. In case of images, this would be the colors in the image.
- *Coding* refers to a representation of symbols as bit patterns. For example, coding a color would be possible by using the bit representation of the numbers specifying the intensity of the color channels.
- *Compression* refers to a special coding for given data such that the total number of bits required to represent the data is minimized.

In the following sections we will explain *Run-length Encoding*, *Operator Encoding* and *Entropy Encoding*.

In the last section we will take a look at the *Wavelet Transform*. Using entropy encoding on wavelet-transformed images allows for very high compression rates. Furthermore, if a slight loss in image quality is acceptable, the compression rate can be increased even more. Also, we can implement progressive transmission of images using this technique.

Note that although we will discuss compression in the context of images, the methods used for compression can be extended to other data as well. Note that in the general definition of data, symbols could represent anything, for example letters of an alphabet (to compress text) or vertex positions and connectivity (for mesh compression).

Special algorithms for mesh compression will be part of the lecture ,,Computer Graphics II“, and are therefore omitted here.

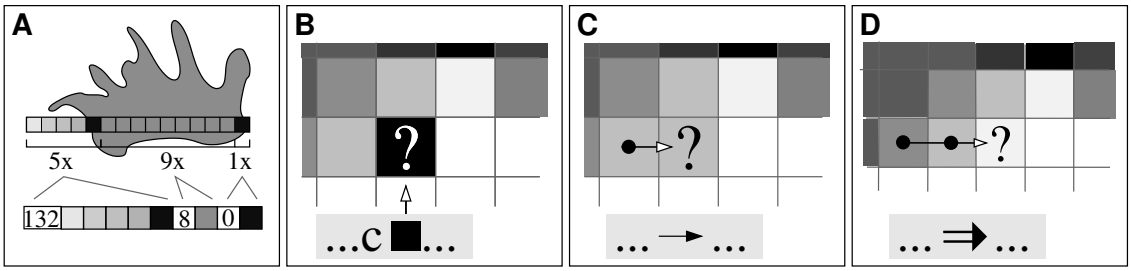


Figure 17.1: Encoding Schemes: (a) run-length encoding, (b)-(e) operator encoding.

17.1 Run Length Encoding

Run length encoding traverses the image scanlines in sequential order, and exploits spatial coherence of pixel colors.

More precisely, a sequence of pixels having the same color is squeezed together into two symbols: the number of repetitions n and the color c. For reasons we will see in a moment, the maximum number of symbols merged in this manner is 128:

$$[n-1, c] \longmapsto [\underbrace{c, c, ..., c}_{n}], \quad 0 < n \leq 128.$$

This scheme gets inefficient if colors are not repeated, but alternate instead. Using the above scheme, we would need to associate with each of these pixels not only its color, but also its repetition count $n = 1$, in effect increasing the amount of data. Therefore, if a sequence of different symbols is encountered, we store the length l of this sequence, followed by the actual symbols. Since 128 numbers are already in use, we encode this by the other 128 numbers available in a byte, so we need one additional symbol for any alternating sequence of at most 128 pixels:

$$[l + 127, c_1, c_2, ..., c_l] \longmapsto [c_1, c_2, ..., c_l], \quad 0 < l \leq 128.$$

A small example of this is shown in Fig. 17.1a.

It makes sense to handle the color channels separately, since the spatial coherence within a color channel is stronger than the spatial coherence in their mixture.

Let us have a look at the compression rates in the worst case as well as in the best case, obtained for a sequence of n colors, each color requiring (for matters of simplicity) one byte.

The worst case occurs when the algorithm cannot squeeze together any symbols, and thus needs to subsequently use the second rule. Therefore, we do not actually compress the image, but add one byte every 128 pixels. Therefore, the image size increases by $\lceil n/128 \rceil$ bytes, representing an overhead of approximately 0.8%.

In the best case, however, which occurs when all pixels are the same in groups of 128, we can map 128 pixels to two bytes, and hence compress the image down to 1/64 of its original size.

The average case will be somewhere in between. Note however that in case of three identical consecutive pixels we start to compress the image already.

17.2 Operator Encoding

One of the problems of run-length encoding is that spatial coherence is exploited in one direction only, i.e. along scanline-order. *Operator encoding* extends this by defining operations allowing for individual computations.

Note that using this scheme is very easy: we just traverse the image in scanline order, and try to find an operator expressing the color of the current pixel in terms of its neighbors. If none is found, we send the color of the current pixel instead, and proceed with the next pixel. The decoder just reads the operators, and reconstructs the image accordingly.

In this section we will present some example operators, however, in a real implementation, other operators might be used. The main point is that each pixel color should be representable based on the operators, without having to specify a new color (except of the first one, of course). Then, we would be able to encode each pixel using one operator, and since the set of operators can be assumed to be much smaller than the set of colors, storing them requires much less memory. Therefore, choosing an efficient set of operators is also vital for this compression technique.

A very basic operator which is necessary in any case is an operator that defines a color to be set, i.e. directly after this operator we send the bit representation of a color, causing the current pixel to be filled with this color (cf. Fig. 17.1b). We shall call this operator "c".

Copy Operator

Another useful operator is the copy operator, which copies a color from one of the already processed neighboring pixels to the current pixel (cf. Fig. 17.1c). Depending on the pixel from which to copy, this leads to a set of operators, which we will denote by arrows: "$\rightarrow$" (copy from left), "$\downarrow$" (copy from above), "$\searrow$" (copy from upper left) and "$\swarrow$" (copy from upper right).

We extend the copy operator by allowing a color offset to be added to the color obtained from the neighboring pixel. We denote this offset by a subscript to the copy operator, for example "$\rightarrow_2$" refers to adding 2 to the pixel color of the left neighbor, and "$\rightarrow_{-1}$" refers to subtracting 1 from it. Note that taking four copy operators, and extending them by an offset between -2 and $+2$, we get a set of 21 operators which can be expressed using 5 bits, much less than we would need to encode a color.

Gradient Operator

We can also define a gradient operator, suitable to encode color gradients. Similar to the copy operator, we specify a direction, but this time choose two pixels in this direction. If a is the color of the direct neighbor and b is two pixels away from the current pixel in the same direction, we now compute the color c of the current pixel by linearly extrapolating the others (cf. Fig. 17.1d):

$$c = a + (a - b) = 2a - b.$$

We shall denote this operator by arrows having two lines. Thus, "$\Rightarrow$" means to fill the current pixel with twice the color of the left neighbor minus the color of the left neighbor's left neighbor. Similarly to how we extended the copy operators, we can also add an offset to the value computed in this way, and denote this offset by a subscript to the operator (e.g. "$\Rightarrow_1$" means to fill the pixel with the "$\Rightarrow$"-operator, and add 1 to the value obtained).

Discussion

The operators we have discussed so far normally will capture a lot of the cases occurring in the image. Note that including offsets from -2 to 2 we have 41 operators in total, requiring six bits. If a color is stored using $3 \cdot 8 = 24$ bits, this achieves an average compression of 75%, and leaves space for even more operators. By reducing the set of operators, we can achieve even higher compression rates.

Note however, the fewer operators, the less cases can be captured, and the more colors need to be sent explicitly using the "c"-operator. Conversely, too many rarely used operators waste space since they require more bits for encoding.

17.3 Entropy Encoding

The *entropy* of a given sequence of symbols yields the number of bits required to encode a symbol in this sequence such that the total length of the sequence gets minimal. We will mathematically formalize the entropy in the next section, for now it is only important to know that any encoding scheme is limited by the entropy of the sequence to be encoded.

Fortunately, based on the mathematical formulation, we can derive properties of the entropy which help us to find minimal codes for a given sequence. *Entropy encoders* try to find such minimal codes.

We will discuss *Huffman coding* here, which achieves an average code length near to the entropy of a sequence. Another method, called *Arithmetic coding*, can be shown to actually achieve the best compression rates, since it reaches the minimum average code length as given by the entropy of the sequence.

17.3.1 Entropy

Let $\{p_0, p_1, ..., p_{n-1}\}$ be a set of symbols, and $S = [s_1, s_2, ..., s_m]$ a sequence of these symbols (i.e. $s_i \in \{p_0, ..., p_{n-1}\}$).

Denoting by n_i the number of occurrences of p_i in S, i.e.

$$n_i = |\{s \in S \,:\, s = p_i\}|, \quad 0 \leq i \leq n-1,$$

and by P_i the occurrence probability of p_i, i.e.

$$P_i = \frac{n_i}{m} \in [0, 1],$$

the *entropy* $E(S)$ of this sequence is defined to be

$$E(S) := -\sum_{i=0}^{n-1} P_i \log_2(P_i) = -\sum_{i=0}^{n-1} \frac{n_i}{m} \log_2\left(\frac{n_i}{m}\right).$$

Note that this value is non-negative since $n_i/m \in [0, 1]$, hence the summands are negative.

Conditions for Maximum Entropy

It is interesting to find conditions for which the entropy of a sequence gets maximal, corresponding to the case where the compression reaches its minimum rate (high average number of bits). Inverting these conditions gives us an idea of what to focus on when designing an entropy encoder.

Since the total probability for any symbol to occur must be one, we directly conclude that

$$\sum_i P_i = 1 \iff P_k = 1 - \sum_{i \neq k} P_i.$$

Now let us take a look at the entropy formula

$$\begin{aligned} E(S) &= -\sum_i P_i \log_2 P_i = -\sum_{i \neq k} P_i \log_2 P_i - P_k \log_2 P_k \\ &= -\sum_{i \neq k} P_i \log_2 P_i - \left(1 - \sum_{i \neq k} P_i\right) \log_2 \left(1 - \sum_{i \neq k} P_i\right). \end{aligned}$$

We want to find the maxima of the entropy, which correspond to the roots of the gradient of $E(S)$, i.e. of all its partial derivatives:

$$\begin{aligned}\frac{\partial}{\partial P_j}E(S) &= -(\log_2 P_j + P_j \cdot \frac{1}{P_j}) + \log_2\left(1 - \sum_{i\neq k} P_i\right) + 1 \\ &= -\log_2(P_j) + \log_2\left(1 - \sum_{i\neq k} P_i\right) \\ &= -\log_2(P_j) + \log_2(P_k)\end{aligned}$$

We want this derivative to become zero, hence

$$\begin{aligned}& -\log_2(P_j) + \log_2(P_k) = 0 \\ \iff\quad & \log_2(P_j) = \log_2(P_k) \\ \iff\quad & P_j = P_k\end{aligned}$$

Since we did not restrict k and j, the maximum entropy is reached if $P_j = P_k$ for any choice of j and k, i.e. if all symbols appear with equal probability:

$$P_0 = P_1 = ... = P_{n-1} = \frac{1}{m}.$$

Therefore, the worst case for entropy encoders is a sequence of symbols in which every symbol occurs with same probability (i.e. if the symbols are uniformly distributed).

Implications for Entropy Encoders

Let us consider encoding the following text:

```
Bla bla bla bla Internet bla bla bla bla bla bla
bla bla Multimedia bla bla bla bla bla bla bla bla
bla bla bla bla bla bla bla Microsoft bla bla
bla bla bla objektorientiert bla bla bla bla bla
bla 3D-Graphik bla bla bla bla bla bla bla bla
bla bla bla bla bla bla E-Commerce bla bla.
```

Using standard 8 bit ASCII-code we would need 2296 bits to encode the sequence. Even if using a code with 5 bits, which would suffice to encode the letters in the text, we would still need 1435 bits. The entropy tells us that we only need 503 bits.

Looking at the text we see that most symbols belong to unimportant text, i.e. to text which is repeated quite often. The important information is contained in very few parts of the text, such as “Multimedia” or “Internet” in the above example.

If we encode frequently used symbols by small bit patterns, and use longer ones for rare symbols, we will get a code yielding the best compression. In the above case, it would be best to encode the letters ,,b“, ,,l“, and “a” using few bits only, since they are used most frequently, and to choose longer bit patterns for more infrequent symbols, like ,,p“, for example. By doing so, every bit in the encoded sequence carries the same amount of information.

Taking a closer look at the entropy, we can see that this is what the entropy does. It gives a particular symbol a bit length according to its occurrence frequency, and sums up these lengths weighted according to their occurrence frequency, therefore yielding the minimum average number

of bits for a symbol. As such, the entropy can be understood as a measure of the amount of information in a text.

Also, since the worst entropy is reached if the symbols are uniformly distributed, it would be good to find an equivalent representation for the signal in which the symbols are not evenly distributed, i.e. where only few symbols make up a large amount of the data.

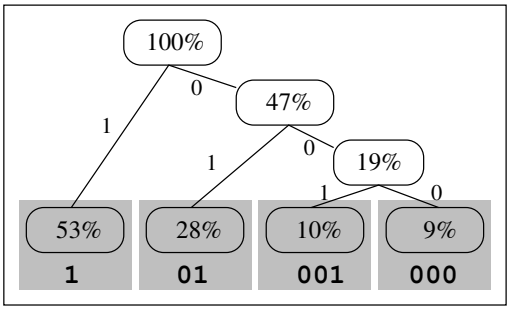
17.3.2 Huffman Coding

The *Huffman Coding* algorithm sorts the symbols by their occurrence probability, and then assigns bit patterns to them as proposed by the entropy.

The first step is to compute the symbols' probabilities and to sort them accordingly. Then, the Huffman algorithm recursively creates a binary tree, with the symbols being the leaf-nodes. With each symbol we associate its occurrence probability. Note that initially each of the leaves represents a binary tree itself, with the root being the leaf.

Now, in each step we join the two trees having roots labelled with the two smallest occurrence probabilities to a new tree by adding a common parent node referencing them. This new node (which is the root node of the new tree) is then labelled with the sum of the occurrence probabilities of its leaves. The process stops when all trees have been integrated into one tree.

The figure on the right shows such a tree. Note that symbols with frequent occurrence have a short path to the root, whereas rare symbols have a long path. Therefore, we assign codes to the symbols corresponding to their path in this tree. Starting at the root node, we add a „1“ to the code anytime we descend into the left subtree, and a „0“ anytime we descend into the right subtree.



Using these codes leads to an average code length near to the entropy of a sequence. For the occurrences $(53\%, 28\%, 10\%, 9\%)$ in the example, a simple scheme represents each symbol using two bits. In contrast to that, using Huffman coding we get a much smaller average bit number:

$$0.53 \cdot 1 + 0.28 \cdot 2 + 0.10 \cdot 3 + 0.09 \cdot 3 = 1.66 \text{ bits per symbol.}$$

The entropy of this sequence yields a minimum of 1.64 bits per symbol, so using the Huffman encoding scheme, we are pretty close to this lower bound.

17.3.3 Image Encoding

Note that when encoding an image, a first idea would be to choose the colors of the image to be the symbols and the sequence corresponding to the scanline ordered pixels. Using this approach would exploit spatial coherence of the pixel colors.

However, recall that we would get the best entropy (i.e. the lowest), if we had a representation of the image resulting in a histogram having only few (but significant) peaks.

In an image we cannot make any assumptions about the occurrence frequencies of the colors. In a balanced image, we would get an uniform color distribution, whereas in others we might indeed get peaks in the histogram.

However, in chapter 15.1 we have found a representation which usually has peaks around zero and descends fast into both directions: the Fourier transform of the signal. Thus it seems better to encode the frequency spectrum of the image using e.g. Huffman coding.

The so-called *Wavelet transform* usually achieves an even better representation of an image. In contrast to the Fourier transform, this transformation decomposes an image into different frequencies, but uses spatial information as well. We will discuss this transformation in the next section.

17.4 Wavelet Transformation

The *Wavelet transform* is, similar to the Fourier transform, a map from spatial domain to frequency domain, more precisely, it decomposes the image successively into high frequencies and low frequencies. However, in contrast to the Fourier transform (cf. Sec. 15.1), spatial information is used as well.

After introducing the wavelet transform, we will discuss the advantages it provides for the compression of images. Finally we will present progressive transmission as another application of the wavelet transform.

17.4.1 From Fourier to Wavelets

In order to motivate the use of the wavelet transform, we will first point out why the Fourier transform is not suitable for signal compression.

Recall from Sec. 15.1 that the Fourier transform decomposes a given signal into sine waves of different frequencies and phases. The problem with these basic waves is that they have infinite support. Consider a signal that is basically smooth but contains a high frequency at just one position. We have to use a high-frequent wave to represent this feature, but this wave also interferes with the smooth parts of the signal. Hence, we need even higher frequencies to correct this.

We can see this problem already in the basic example of Fig. 15.1a: Although the piecewise constant rectangle signal contains just two frequencies (0 and ∞), we have to use all real-valued frequencies for representing it. In the case of images it is in fact very likely that some parts of the image contain high frequencies (e.g. sharp color edges), while other parts may be homogeneous.

The wavelet transform therefore uses basic waves that correspond to just one period of a sine wave, hence having finite compact support. This results in an orthogonal basis of the function space L^2 consisting of all scaled and shifted copies of one basic wave ψ (the *mother wavelet*):

$$\left\{ \psi_{t,s}(x) = \psi\left(\frac{x-t}{s}\right),\ \ t, s \in \mathbb{R} \right\},$$

where s controls the scaling (frequency) and t the local position. This approach has the advantage that it can resolve high frequencies locally. The wavelet transform of a signal $f(x)$ assigns to each pair (s,t) the amount of the frequency s at position t in f:

$$(s,t) \longmapsto \langle f, \psi_{t,s} \rangle = \int_{-\infty}^{\infty} f(x)\, \overline{\psi\left(\frac{x-t}{s}\right)}\, dx.$$

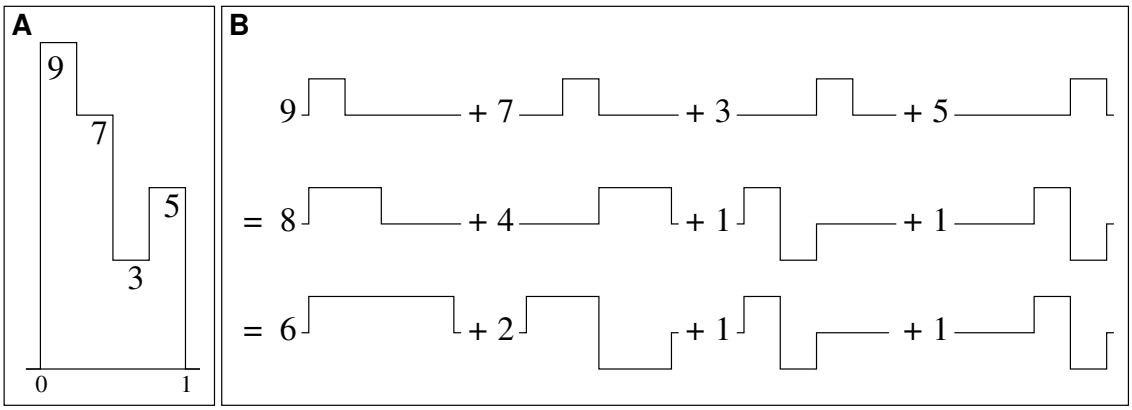


Figure 17.2: Haar Wavelet Decomposition: (a) the original signal, (b) successively decomposition using scaled box functions ϕ_i^j and wavelets ψ_i^j.

17.4.2 Haar Wavelets

Since images can be considered piecewise constant functions it makes sense to use wavelets that are also piecewise constant. Therefore we will restrict to the so-called *Haar wavelets* from now on.

These are defined by the scaled box functions

$$\begin{aligned} \phi(x) &= \begin{cases} 1, & 0 \le x < 1 \\ 0, & \text{otherwise} \end{cases} \\ \phi_i^j(x) &= \phi\left(2^j x - i\right), \end{aligned}$$

and the wavelet functions

$$\begin{aligned} \psi(x) &= \begin{cases} 1, & 0 \le x < \frac{1}{2} \\ -1, & \frac{1}{2} \le x < 1 \\ 0, & \text{otherwise} \end{cases} \\ \psi_i^j(x) &= \psi\left(2^j x - i\right), \end{aligned}$$

The box functions ϕ_i^j build a nested set of spaces, since every function represented by $\{\phi_i^k,\ i \in \mathbb{Z}\}$ can also be represented by $\{\phi_i^{k+1},\ i \in \mathbb{Z}\}$. The central point is that the wavelet functions complement the box functions to build a basis of the next level space, i.e. $\{\phi_i^k,\ i \in \mathbb{Z}\} \cup \{\psi_i^k,\ i \in \mathbb{Z}\}$ build a basis of $\{\phi_i^{k+1},\ i \in \mathbb{Z}\}$.

Now let us consider a signal to be a piecewise constant function over $[0, 1]$, as depicted in Fig. 17.2a. If the signal contains 2^n values it can be represented by the box functions (cf. Fig. 17.2b, first row):

$$\left\{\phi_0^n, \ldots, \phi_{2^n-1}^n\right\}.$$

But we can also represent this signal using the box functions and wavelets of level $n-1$, thereby decomposing the signal into two low frequencies and two high frequencies (cf. Fig. 17.2b, second row).

Iterating the same technique, we can represent the signal by ϕ_0^0 (representing the global average or direct current), plus a set of 2^{n-1} wavelets, corresponding to the higher frequencies (cf. Fig. 17.2b, third row). The Haar wavelet transform can therefore be considered as a basis change from spatial domain to *spatial frequency domain*

$$\{\phi_0^n, \ldots, \phi_{2^n-1}^n\} \longmapsto \phi_0^0 \cup \psi_0^0 \cup \left\{\psi_0^1, \psi_1^1\right\} \cup \ldots \cup \left\{\psi_0^{n-1}, \ldots, \psi_{2^{n-1}-1}^{n-1}\right\}.$$

17.4.3 Haar Wavelet Analysis

In practice it may be too expensive to compute the wavelet coefficients by orthogonal projection, i.e. by evaluating

$$\left\langle f, \psi_i^j \right\rangle = \int_{-\infty}^{\infty} f(x)\ \psi_i^j(x)\, dx.$$

We will therefore derive an efficient hierarchical scheme for computing the Haar wavelet decomposition in this section.

The wavelet transformation is applied to an array of 2^n values, and decomposes it into another array of 2^n values. In the resulting array, the first value represents the direct current of the values (i.e. the overall average), whereas the other values contain higher frequencies of the image. The decomposition is recursively computed as follows.

The given array of values $V = [v_1, \ldots, v_{2^n}]$ is first decomposed into pairs of two consecutive values $(v_1, v_2), (v_3, v_4), \ldots, (v_{2^n-1}, v_{2^n})$. For each of these pairs (v_{2i-1}, v_{2i}) we compute its average (corresponding to a low-pass filter),

$$l_i = \frac{v_{2i-1} + v_{2i}}{2},$$

and its difference (corresponding to a high-pass filter),

$$h_i = \frac{v_{2i} - v_{2i-1}}{2}.$$

The resulting values are rearranged to build a new array

$$V' = [l_1, l_2, \ldots, l_{2^{n-1}}, h_1, h_2, \ldots, h_{2^{n-1}}].$$

We continue the wavelet transform on the first half of the new array, i.e. we transform

$$[l_1, l_2, \ldots, l_{2^{n-1}}].$$

This process is repeated until we have reached an array of length one.

Example Let us have a look at an example, which should clarify the transformation. Let the array be $V = [3, 5, 2, 2, 3, 7, 7, 11]$. Using averaging and differencing on the pairs $(3, 5)$, $(2, 2)$, $(3, 7)$ and $(7, 11)$, we get the following low and high frequencies:

$$L = [4, 2, 5, 9] \text{ and } H = [1, 0, 2, 2]$$

and therefore the result of the first transformation level is

$$V' = [4, 2, 5, 9, 1, 0, 2, 2].$$

Now, the new input array is $L = [4, 2, 5, 9]$ with pairs $(4, 2)$ and $(5, 9)$, and therefore $L' = [3, 7]$ and $H' = [-1, 2]$, hence our result of the second transformation is

$$V'' = [3, 7, -1, 2, 1, 0, 2, 2].$$

Again, we transform the low frequencies, $L' = [3, 7]$, into a low frequency array $L'' = [5]$ and a high frequency array $H'' = [2]$, and the final result is

$$V''' = [5, 2, -1, 2, 1, 0, 2, 2].$$

17.4.4 Haar Wavelet Synthesis

The inverse of the wavelet analysis is the synthesis that reconstructs the original signal from the decomposed one. Again a hierarchical scheme can be used, resulting in an efficient algorithm.

For the composition we have to do the inverse calculations of the decomposition. A pair of values v_{2i-1} and v_{2i} can be reconstructed from their average l_i and difference h_i according to the formula

$$\begin{aligned} v_{2i-1} &= l_i - h_i \\ v_{2i} &= h_i + l_i. \end{aligned}$$

Example Let us reconstruct the signal of the last example again. We are given the decomposed values $V''' = [5, 2, -1, 2, 1, 0, 2, 2]$. We now start on the lowest level, i.e. the sub array $[5, 2]$, corresponding to a low and a high frequency. Using sum and difference, we obtain

$$v''_1 = 5 - 2 = 3 \quad \text{and} \quad v''_2 = 5 + 2 = 7,$$

and hence the new array is

$$V'' = [3, 7, -1, 2, 1, 0, 2, 2].$$

Next we consider the first four values,

$$v''_1 = 3 - (-1) = 4, \quad v''_2 = 3 + (-1) = 2, \quad v''_3 = 7 - 2 = 5, \quad v''_4 = 7 + 2 = 9.$$

Thus, the array of the next higher level is

$$V' = [4, 2, 5, 9, 1, 0, 2, 2],$$

and after another step we finally obtain the solution

$$V = [3, 5, 2, 2, 3, 7, 7, 11].$$

17.4.5 2D Haar Wavelet Transform

For the Haar wavelet transform of images we have to transform a two-dimensional array of values. Although we could connect all scanlines to build one large vector of $w \cdot h$ values, this would discard spatial coherence of neighboring scanlines.

Instead, we alternatingly apply one step of the wavelet transform on the columns, and then one step on the rows. Fig. 17.3 shows this technique for an example image. Note that the histogram gets narrower in each of the steps as high frequencies are small (small values corresponding to gray color).

Since the histogram now has a strong peak (cf. Fig. 17.3), entropy encoding will yield much more compact results using this representation.

17.4.6 Compression

The last sections showed that a given signal can be decomposed into its wavelet coefficients, and that the original signal can be reconstructed from these coefficients again.

Since the number of wavelet coefficients equals the number of values in the signal, this alone does not lead to a compression effect. But since most of the coefficients represent the amount of high frequencies in the signal, these values are generally very small.

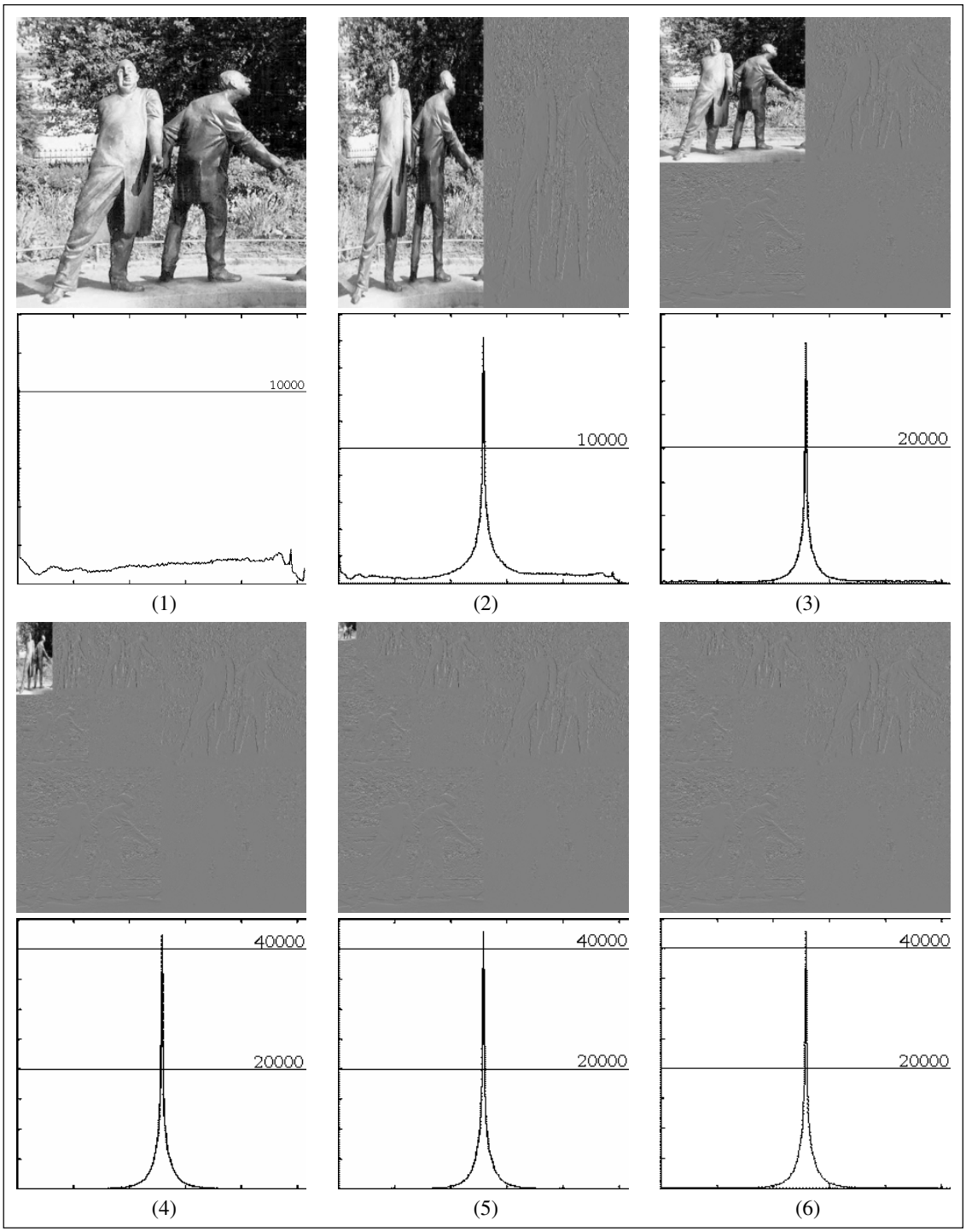


Figure 17.3: 2D Wavelet Decomposition: The original (a) and the wavelet decomposition after one step (b) two steps (c), 5 steps (d), 8 steps (e) and the final result after 11 steps (f) are shown together with their histograms.

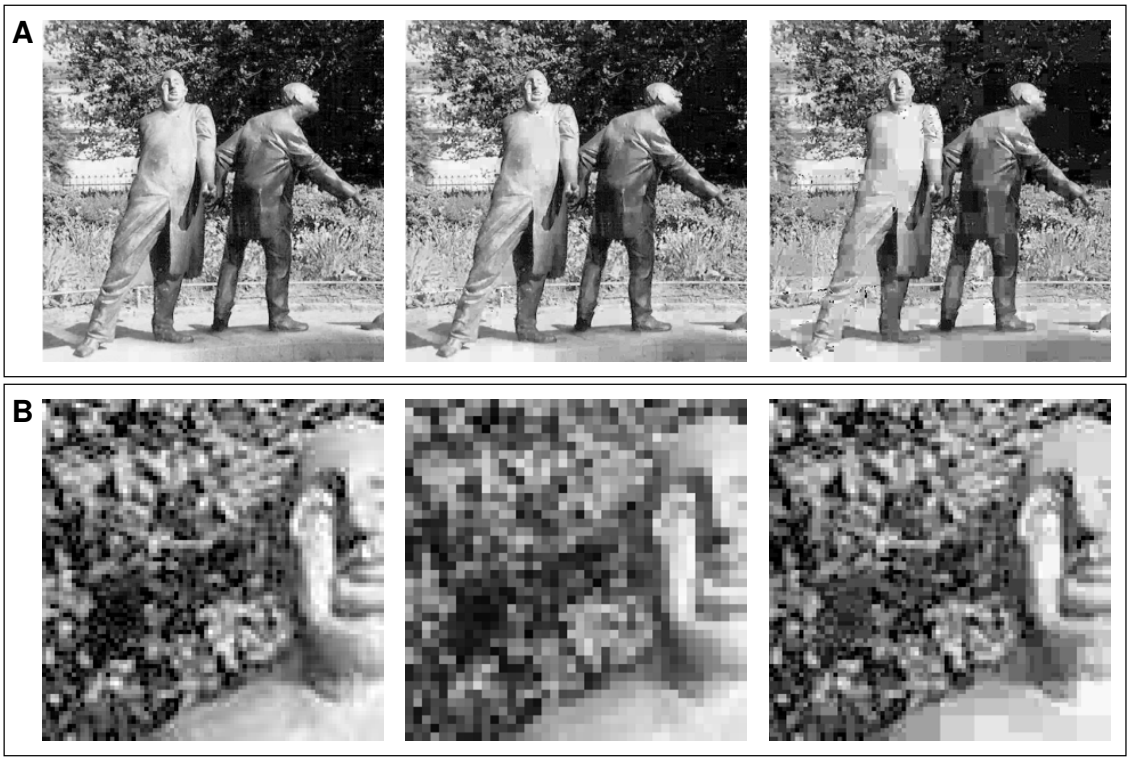


Figure 17.4: Lossy Compression: The upper row shows lossy wavelet compression to 75%, 50% and 25% of the original size. In the lower row the original image is compared to a sub-sampled version and the wavelet representation, both having 25% of the original size.

Hence, the histogram of the wavelet coefficients will have a peak around zero and will drop down very fast for higher values. This is exactly the case in which entropy encoding leads to very good results. Consequently, storing the entropy-coded wavelet coefficients instead of the original color values will result in much less space consumption.

We can further reduce the space consumption of an image by slightly reducing the image quality during compression. Since then the original image cannot be reconstructed exactly, this is referred to as *lossy compression*.

In a naive approach, we would simply reduce the number of colors in the image, or its resolution. In both cases, image quality might be reduced drastically. For a reduction to 25% of the original size, we have to reduce the resolution by a factor of two in both dimensions.

The wavelet transform enables for much better results in this setting. Up to now we just changed the representation of the image to get a more compact entropy encoding. However, note that after the wavelet transform small values represent small color differences instead of dark colors. Thus, clamping small values to zero does not degrade image quality substantially. However, it leads to an even more intense peak at zero, resulting in an even higher compression rate.

Fig. 17.4 shows the image quality after reducing the size of the image to 75%, 50% and 25%. For the last compression step, the results of naive resolution reduction and wavelet compression are compared to the original image. Note that the quality of the wavelet image is much higher than the one we obtained with resolution reduction.

However, in homogeneous regions of the image the lossy wavelet compression generates blocks (i.e. square-shaped regions of constant color). The reason is that in homogeneous areas high frequencies are very small, and that clamping them to zero results in blocks of constant color.

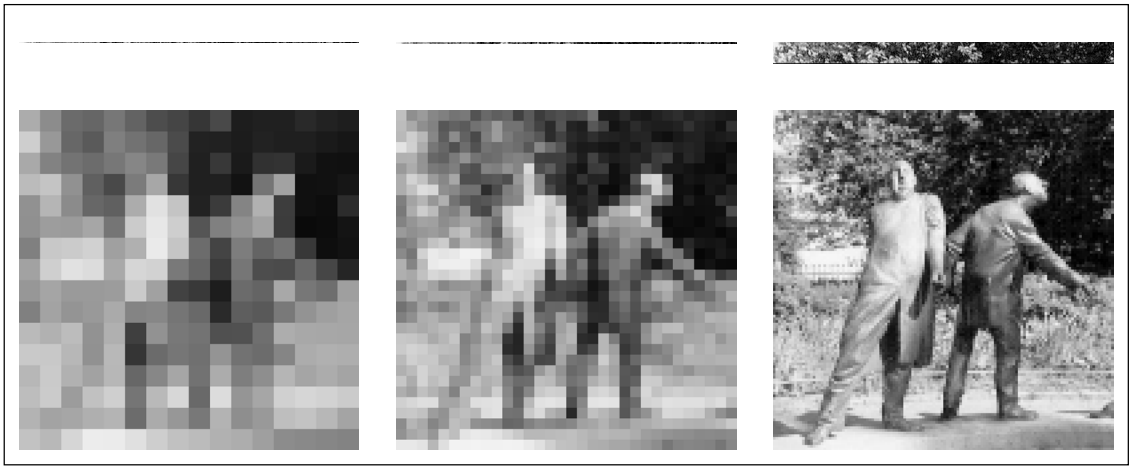


Figure 17.5: Progressive Transmission. The upper row shows the sequential transmission, the lower row progressive transmission. The pictures (left to right) show the result after the data corresponding to one, two and thirty-two rows of the image were transmitted.

17.4.7 Progressive Transmission

Compression can also be used to reduce bandwidth requirements when sending an image over data channels, like e.g. the internet. Since compression reduces the amount of data that has to be transmitted in order to rebuild the image on the receiver-side, less time is needed to receive the image.

However, when sending compressed images scanline by scanline, the image is reconstructed in this order as well. As a consequence, we see only the part of the image that has been received already. Although the received part is in full quality, this behavior is not desirable: we would like to get an impression of the whole image first, and then increase the detail level step by step. In this way, we can stop transmission when we got the required detail or when we realize that we are in fact not interested in the image at all.

Progressive transmission does this by sending data ordered by their relevance. Thus, on the receiver side, we first get a very undetailed but complete image, and while more and more data is retrieved, more and more detail is added until the image is completely rebuilt in full detail.

The wavelet transform performs exactly this ordering of information. It decomposed an image into global information (low frequencies) and detail information (high frequencies).

Consider what happens if we scale the image to its original size after each wavelet reconstruction step. In the first step (after one value is received), we see the whole image area filled with the global average of its colors. Every further reconstruction step (in x and y direction) doubles the resolution and adds more and more details to the image.

Fig. 17.5 compares this kind of progressive transmission to sequential transmission. Each pair of images shows the results after the same amount of data has been received. Using progressive transmission we have a good impression of the image already when the sequential transmission has only reconstructed the first few lines of the image.

Appendix A

Linear Algebra Basics

In this computer graphics introduction most of the mathematical tools we need are direct applications of basic linear algebra. For instance, we will be dealing with the Euclidean vector spaces $\mathbb{R}^2$ or $\mathbb{R}^3$ and with the affine space $\mathbb{R}^4$. Most transformations we perform on geometric objects in order to render them into an image are simple vector space operations. Therefore we will repeat the basics of linear algebra and analytical geometry.

Vector Spaces

A *vector space* or *linear space* over $\mathbb{R}$ is a set of vectors V providing the two operations

$$\begin{aligned} + &: V \times V \to V, \qquad (v, w) \mapsto v + w \\ \cdot &: \mathbb{R} \times V \to V, \qquad (s, v) \mapsto s \cdot v \end{aligned}$$

such that for all vectors $v, w \in V$ and scalars $a, b \in \mathbb{R}$ the following holds:

- $(V, +)$ is a commutative group.
- $a \cdot (v + w) = a \cdot v + a \cdot w$
- $(a + b) \cdot v = a \cdot v + b \cdot v$
- $a \cdot (b \cdot v) = (a \cdot b) \cdot v$
- $1 \cdot v = v$

Vector Combinations

A sum $\sum_i \alpha_i \cdot v_i$ with $\alpha_i \in \mathbb{R}$ and $v_i \in V$ is called a *linear combination* of the vectors v_i. The set of vectors is called *linearly independent* if any of them cannot be represented by a non-trivial linear combination of the others. A linear combination $\sum_i \alpha_i v_i$ is called *affine combination* if the α_i sum to one, i.e. $\sum_i \alpha_i = 1$. If, in addition, all the α_i are non-negative this is a *convex combination*.

Basis

The span $\langle v_1, \ldots, v_n \rangle$ of a set of vectors $v_i \in V$ is the linear subspace $\{\sum_{i=1}^n \alpha_i v_i \,,\ \alpha_i \in \mathbb{R}\}$ of all linear combinations of these vectors. A set of linearly independent vectors $\mathcal{B} = \{b_1, \ldots, b_n\}$ spanning a (sub)space V is called a *basis* of this space. Since any vector $v \in V$ can be uniquely

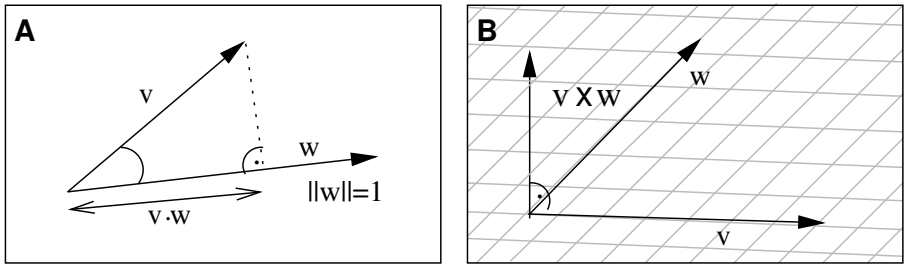


Figure A.1: Vector products: (a) scalar product, (b) cross product.

represented by a linear combination $v = \sum_{i=1}^{n} \alpha_i b_i$, the n-tuple $(\alpha_1, \ldots, \alpha_n)^T =: v_{\mathcal{B}}$ is referred to as the coordinates of v w.r.t. the basis $\mathcal{B}$.

The *standard basis* $\mathcal{E} = \{e_1, \ldots, e_n\}$ is built from the unit vectors $e_i := (\overbrace{0, \ldots, 0}^{i-1}, 1, \overbrace{0, \ldots, 0}^{n-i})^T$. We use this basis as the default one (i.e. $v = v_{\mathcal{E}}$).

Vector Products and Norms

A map

$$\langle ., . \rangle \, : \, V \times V \to \mathbb{R} \, , \; (v, w) \mapsto \langle v, w \rangle$$

that is bilinear (i.e. linear in both arguments) as well as symmetric (i.e. $\langle v, w \rangle = \langle w, v \rangle$) is called *scalar product*. A vector space providing a scalar product is called *Euclidean* space, e.g. the Euclidean vector space $\mathbb{R}^n$ with the canonical scalar product

$$\langle v, w \rangle := \sum_{i=1}^{n} v_i \cdot w_i.$$

A map $\|.\| : V \to \mathbb{R} \, , \; v \mapsto \|v\|$ is called a *norm* if

- $\|\alpha \cdot v\| = |\alpha| \cdot \|v\|$
- $\|v + w\| \leq \|v\| + \|w\|$
- $\|v\| = 0 \Leftrightarrow v = 0.$

A vector v is called *normalized* if its norm $\|v\|$ equals 1. Any vector can be normalized by dividing it by its norm $v \mapsto v / \|v\|$.

A scalar product can be used to induce a norm by $\|v\| := \sqrt{\langle v, v \rangle}$. A Euclidean space with a norm defined by its scalar product is called a *Hilbert space*, e.g. the Euclidean space $\mathbb{R}^n$ with the *Euclidean norm*

$$\|v\| := \|v\|_2 := \sqrt{\langle v, v \rangle} = \sqrt{\sum_{i=1}^{n} v_i^2},$$

evaluating to the length of the vector v.

Combining scalar product and norm we can derive the following fact for the space $\mathbb{R}^n$:

$$\langle v, w \rangle = \|v\| \, \|w\| \cos \left(\angle (v, w) \right)$$

Therefore, the scalar product provides a way to measure angles between vectors, especially $\langle v, w \rangle = 0 \Leftrightarrow v \perp w$ (cf. Fig. A.1a).

A map specially defined for the space $\mathbb{R}^3$ is the *cross product*

$$\times : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{R}\,,\ (v, w) \mapsto v \times w := (v_2 w_3 - v_3 w_2, v_3 w_1 - v_1 w_3, v_1 w_2 - v_2 w_1)^T\,.$$

Geometrically, the cross product yields a vector orthogonal to both v and w with the length $\|v\|\,\|w\|\sin(\angle(v, w))$ (i.e. proportional to the area of the triangle spanned by v and w, cf. Fig. A.1b).

Matrices

A *matrix* is a representation for a linear operator acting on vectors and will be denoted by capital letters. A matrix $M \in \mathbb{R}^{m\times n}$ has m rows and n columns. Using this notation a vector $v \in \mathbb{R}^n$ will be regarded as a $n \times 1$ matrix. Elements of a matrix M are specified by

- $M_{i,j}$: the entry in the i'th row and j'th column.
- $M_{i,.}$: the i'th row.
- $M_{.,j}$: the j'th column.

The multiplication of two matrices (or a matrix and a vector) $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{n\times o}$ yields a matrix $C \in \mathbb{R}^{m\times o}$ defined by

$$C_{k,l} = \sum_{i=1}^{n} A_{k,i} \cdot B_{i,l}.$$

A special matrix is the *identity matrix* I which is a square diagonal matrix $diag(1, \ldots, 1)$ with ones on the diagonal and zeros otherwise. Multiplication with I yields the identity, i.e. $M \cdot I = I \cdot M = M$ and $I \cdot v = v$.

Bibliography

[Blinn and Newell, 1976] Blinn, J. and Newell, M. E. (1976). Texture and Reflection in Computer generated images. *Communications of the ACM*, 19(10):542–547.

[Coxeter, 1969] Coxeter, H. S. M. (1969). *Introduction to Geometry.* Wiley Classics Library. John Wiley & Sons, 2nd edition.

[Crow, 1977] Crow, F. C. (1977). Shadow Algorithms for Computer Graphics. In *Siggraph 77 conference proceedings.*

[Dalheimer, 2002] Dalheimer, M. K. (2002). *Programming with Qt.* O'Reilly & Associates, 2nd edition.

[Farin, 1997] Farin, G. (1997). *Curves and Surfaces for Computer Aided Geometric Design.* Academic Press, 4th edition.

[Foley et al., 2001] Foley, van Dam, Feiner, and Hughes (2001). *Computer Graphics: Principles and Practice, 2nd edition.* The Systems Programming Series. Addison-Wesley.

[Glassner, 1993] Glassner, A. S., editor (1993). *An Introduction to Ray Tracing.* Academic Press.

[Golub and van Loan, 1996] Golub, G. H. and van Loan, C. F. (1996). *Matrix Computations.* Johns Hopkins University Press.

[Kobbelt et al., 2001] Kobbelt, L., Botsch, M., Schwanecke, U., and Seidel, H.-P. (2001). Feature Sensitive Surface Extraction from Volume Data. In *Siggraph 01 Conference Proceedings*, pages 57–66.

[Lacroute and Levoy, 1994] Lacroute, P. and Levoy, M. (1994). Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation. In *Siggraph 94 Conference Proceedings.*

[Levoy, 1988] Levoy, M. (1988). Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8(3):29–37.

[Levoy, 1990] Levoy, M. (1990). Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics*, 9(3):245–261.

[Lorensen and Cline, 1987] Lorensen, W. E. and Cline, H. E. (1987). Marching Cubes: a High Resolution 3D Surface Construction Algorithm. In *Siggraph 87 Conference Proceedings)*, volume 21, pages 163–170.

[Miller and Hoffman, 1984] Miller, G. S. and Hoffman, C. R. (1984). Illumination and Reflection Maps: Simulated Objects in Simulated and Real Environments. In *Siggraph 84 course notes.*

[Prautzsch et al., 2002] Prautzsch, H., Boehm, W., and Paluszny, M. (2002). *Bezier and B-Spline Techniques.* Springer.

[Press et al., 2002] Press, W. H., Teukolsky, S. A., Vetterlink, W. T., and Flannery, B. P. (2002). *Numerical Recipes in C++: The Art of Scientific Computing*. Cambridge University Press, 2nd edition.

[Sar Dessai, 2003] Sar Dessai, S. (2003). Vorlesungsmitschrift Computer Graphics II. URL `http://www.sandip.de/showpage.php?page=downloads`.

[Sederberg and Parry, 1986] Sederberg, T. W. and Parry, S. R. (1986). Freeform Deformation of Solid Geometric Models. In *SIGGRAPH 86 Conference Proceedings*, pages 151–159.

[Silicon Graphics, 2003] Silicon Graphics (2003). OpenGL Extension Registry. URL `http://oss.sgi.com/projects/ogl-sample/registry/`.

[Stamminger and Drettakis, 2002] Stamminger, M. and Drettakis, G. (2002). Perspective Shadow Maps. In *Siggraph 02 conference proceedings*, pages 557–563.

[Watt, 1999] Watt, A. H. (1999). *3D Computer Graphics*. Addison Wesley, 3rd edition.

[Wernecke, 1994] Wernecke, J. (1994). *The Inventor Mentor*. Addison Wesley, 2nd edition.

[Westermann and Ertl, 1998] Westermann, R. and Ertl, T. (1998). Efficiently using Graphics Hardware in Volume Rendering Applications. In *Siggraph 98 conference proceedings*.

[Westover, 1990] Westover, L. (1990). Footprint Evaluation for Volume Rendering. In *Siggrraph 90 proceedings*, pages 367–376.

[Williams, 1978] Williams, L. (1978). Casting Curved Shadows on Curved Surfaces. In *Siggraph 78 conference proceedings*.

[Woo et al., 1999] Woo, M., Neider, J., Davis, T., and Shreiner, D. (1999). *OpenGL Programming Guide*. Addison Wesley, 3rd edition.

Index